Olaf Manz

Fehlerkorrigierende Codes

Konstruieren, Anwenden, Decodieren

Springer Vieweg

Fehlerkorrigierende Codes

Olaf Manz

Fehlerkorrigierende Codes

Konstruieren, Anwenden, Decodieren

Springer Vieweg

Olaf Manz
Worms, Deutschland

ISBN 978-3-658-14651-1 ISBN 978-3-658-14652-8 (eBook)
DOI 10.1007/978-3-658-14652-8

Springer Vieweg

Planung: Ulrike Schmickler-Hirzebruch

Gedruckt auf säurefreiem und chlorfrei gebleichtem Papier.

Springer Vieweg ist Teil von Springer Nature
Die eingetragene Gesellschaft ist Springer Fachmedien Wiesbaden GmbH
Die Anschrift der Gesellschaft ist: Abraham-Lincoln-Strasse 46, 65189 Wiesbaden, Germany

Vorwort

Wenn man sich in der Literatur zum Thema **fehlerkorrigierende Codes** umsieht, so findet man inhaltlich zwei Kategorien an Büchern und Monografien.

Die einen – und das sind mit Abstand die meisten – gehen das Thema unter dem Titel **Codierungstheorie** eher wissenschaftlich an. Dabei handelt es sich meist um Lehr- und Fachbücher sowie Vorlesungsskripte, die ausgehend vom mathematischen Umfeld die Theorie beim Codieren in den Vordergrund stellen. Als Basis für Vorlesungen oder Seminare muss es auch primäres Ziel sein, den Studenten wissenschaftliches Arbeiten nahezubringen und sie zumindest punktuell an Bereiche der aktuellen Forschung heranzuführen. Gleiches trifft zu für Arbeitsgruppen an Instituten und Hochschulen. Praxisanwendungen sind dabei eher Randthemen und werden manchmal quasi im Vorbeigehen abgehandelt.

Die andere Kategorie zielt eher auf allgemeinverständliches Niveau ab. Das geht in diesem Falle auch gut, da man den wesentlichen Zweck fehlerkorrigierender Codes – nämlich Datenredundanz zur Korrektur von Übertragungsfehlern – auch Laien leicht nahebringen kann. Zum Verständnis genügen dabei einfache Beispiele, die Zahl der Praxisanwendungen ist schier unerschöpflich, und auch die geschichtlichen Aspekte der Codierungstheorie lesen sich ganz spannend. Die Mathematik dahinter bleibt aber in aller Regel verborgen.

Ziel der vorliegenden Ausarbeitung ist ein Spagat zwischen beiden. Fakt ist nämlich, dass man in der Codierungstheorie schon mit vergleichsweise wenig mathematischen Mitteln recht tief in die Materie eindringen kann. Tief eindringen ist dabei nicht im Sinne von Forschung und theoretischer Resultate gemeint, sondern im Sinne der fehlerkorrigierenden Codes selbst – diese stehen also im Mittelpunkt.

- Wir werden dabei berühmten Familien von Codes begegnen, z.B. den **Hamming**-Codes, **Golay**-Codes, **Reed-Muller**-Codes, **Reed-Solomon**-Codes, **BCH**-Codes und **Turbo**-Codes und noch einigen anderen mehr. Dabei konzentrieren wir uns jeweils auf die Art und Weise, wie sie konstruiert werden und welche Gütekriterien sie besitzen. Dazu werden wir etwas Mathematik in die Hand nehmen müssen – sowohl zum konzeptionellen Verständnis als auch für die ein- oder andere formale Herleitung. Wir bauen dies aber stückweise auf – auch den jeweils notwendigen mathematischen Hintergrund – und testen unsere Ergebnisse stets an konkreten Beispielen.

- Spektakulär sind auch die Verfahren, wie man beim Empfang die aufgetretenen Fehler korrigiert, oder sie zumindest erkennt. Man nennt diesen Vorgang **Decodieren** und er erfordert mindestens genauso viel Kreativität wie das Auffinden von geeigneten Codes selbst. Diesem anwendungsorientierten Thema, was manchmal etwas zu kurz kommt, wollen wir uns sehr intensiv widmen. Stichworte sind dabei unter anderem **Syndrom**-, **Majority-Logic**-, **Peterson-Gorenstein-Zierler**-, **Berlekamp**-**Massey**- und **Viterbi**-Algorithmus.
- Neben den Codes selbst stehen natürlich die vielen Anwendungen in diversen Bereichen der Datenübertragung im Zentrum des Interesses. Diese reichen vom alltäglichen Umgang mit **Artikeln** und **Euro-Scheinen**, über den **Mobilfunk**, das **Digitalfernsehen** und die **Satellitennavigation** bis hin zu **Datenautobahnen** und speziell **Internet**. Faszinierend sind sicherlich die Anwendungen in der **Raumfahrt**, die entsprechend umfänglich behandelt werden. Nicht zu vergessen müssen Daten auch beim Speichern bzw. Auslesen aus Speichermedien gegen Fehler geschützt sein. Dies bringt uns zu Themen wie **Audio-CD** und **Daten-DVD**, **Festplatten** und **Speicherchips** sowie zu den heute allgegenwärtigen **2D-Bar-Codes**. Wir schauen uns dabei an, welche Codierverfahren wie und in welcher Konstellation jeweils zum Einsatz kommen.
- Last but not least ist eines schon bei der Aufzählung der Codes und der Algorithmen klar geworden: Die Entwicklung in der Codierungstheorie ist eng mit Menschen verknüpft. Beginnend mit **Claude Shannon's** berühmten **Kanalcodierungssatz** aus dem Jahr 1948 orientiert sich deshalb unsere Darstellung an all den weiteren Meilensteinen, und unternimmt damit den Versuch, ein klein wenig auch die **Geschichte der Codierungstheorie** zu erzählen.

Zielgruppe dieser Ausarbeitung sind grundsätzlich alle, die sich für dieses Thema begeistern können. Das Ganze erfordert nur relativ wenig (Oberstufen-)Mathematik. Wünschenswert wäre eine gewisse Vertrautheit mit Vektorrechnung, linearen (Euklidischen) Räumen und Polynomen. Auf Matrix-Kalkül wird gänzlich verzichtet, Vorkenntnisse in Wahrscheinlichkeitsrechnung sind nur sehr rudimentär notwendig. Allgemeine Herleitungen werden an geeigneten Stellen eingeflochten, besonderer Wert wird hingegen gelegt auf die Plausibilisierung der Zusammenhänge und vor allem auf viele konkrete Beispiele.

Stürzen wir uns also ins Abenteuer – dabei viel Spaß.

Inhaltsverzeichnis

Eingangsbeispiele und Blockcodes

1

Was heißt eigentlich *Codieren*? Wir machen uns zunächst einmal klar, um was es beim **Kanalcodieren** (oder kurz **Codieren**) geht, nämlich zu der eigentlichen Information redundante Daten hinzuzufügen, um Fehler bei der Datenübertragung oder beim Datenauslesen aus einem Speicher selbstständig erkennen oder sogar korrigieren zu können. Dies ermöglicht uns auch gleichzeitig eine klare Abgrenzung zum **Quellencodieren** (optimale, komprimierte Digitalisierung einer Information) und zum **Chiffrieren** (Verschlüsselung einer Information zum Schutz gegen unbefugtes Abhören und Verändern).

Von Parität, Büchern und Euros Wir tasten uns langsam an das Thema heran. Den **Paritätsprüfungscode**, bei dem man einer Bitfolge ein weiteres Bit anfügt, um die Anzahl der Einsen gerade zu machen, kennt vermutlich jeder. Wenn dabei ein Fehler auftritt, kann man dies an der fehlenden Parität erkennen. Wenn man dagegen eine Nachricht mehrfach wiederholt (sog. **Wiederholungscode**), dann ist das zwar sehr sicher, aber auch sehr redundant. Konkrete Beispiele aus dem Alltag wie **Buchnummer**, **Euroseriennumme**r und **Wertpapierkennnummer** (naja, nicht ganz so alltäglich) machen das Ganze noch etwas transparenter.

Blockcodes – Hamming-Abstand und Fehlerkorrektur Nach unserem eher intuitiven Einstieg ist es jetzt Zeit, die wichtigsten Grundbegriffe der sog. **Blockcodes** zusammenzustellen. Ein Blockcode ist eine Teilmenge aller n-Tupel über einem endlichen Alphabet A (sog. **Codewörter**), wobei das **binäre Alphabet** $\{0, 1\}$ mit Abstand das wichtigste sein wird. Zentraler Begriff ist der sog. **Hamming-Abstand**, d. h. die Anzahl der Positionen, an denen sich zwei n-Tupel unterscheiden, und der daraus abgeleitete **Minimalabstand** für den gesamten Code. Mit diesem Abstandsbegriff, der so rein gar nichts mit dem gewohnten euklidischen Abstand zu tun zu haben scheint, kann man dennoch ganz gut arbeiten. Zum Beispiel kann man **Kugeln vom Radius r um ein Codewort** betrachten, aus denen man recht einfach die **Fehlererkennungs- und Fehlerkorrekturkapazität** eines Blockcodes ableiten kann. Grob gesprochen gilt: Je größer der Minimalabstand, umso besser ist

O. Manz, *Fehlerkorrigierende Codes*, DOI 10.1007/978-3-658-14652-8_1

der Code hinsichtlich Fehlerbehandlung. Also sollten wir uns nach Codes mit möglichst großem Minimalabstand umsehen, aber selbstredend auch mit möglichst wenig redundanter Information.

Von Shannon bis zu Artikel- und Kontonummern Die *Botschaft* des **Kanalcodierungssatzes** von **Claude Shannon** aus dem Jahr 1948 besagt, dass man – unter Berücksichtigung der Kapazität eines gegebenen Kanals – beliebig gute Codes konstruieren kann. Dies macht uns grundsätzlich optimistisch bei unserer Suche. Leider sagt uns Shannon nur *dass*, aber nicht *wie*. Wir starten einen ersten allgemeinen, aber leider noch *unbeholfenen* Versuch mit sog. **optimalen Codes** und der **Gilbert-Schranke**. Dann lassen wir uns doch lieber inspirieren von zwei etwas komplexeren Praxisanwendungen, der **IBAN** (**Kontonummer**) und der **EAN** (**Artikelnummer**). Dabei lernen wir auch gleich die Struktur der **Barcode**darstellung der EAN kennen.

1.1 Was heißt eigentlich *Codieren*?

1.1.1 Datenübertragung

Im heutigen Informationszeitalter werden permanent sowohl in der Wirtschaft als auch im Privaten unvorstellbare Mengen von Informationen zwischen Sendern und Empfängern ausgetauscht. Dies kann sowohl über Online-Kanäle als auch über Speichermedien erfolgen. Dabei gibt es grundsätzlich drei Anforderungen zu beachten, wie in Abb. 1.1 dargestellt ist. Die Information sollte

- möglichst platzsparend und komprimiert, aber dennoch *gut lesbar* übertragen bzw. gespeichert werden,

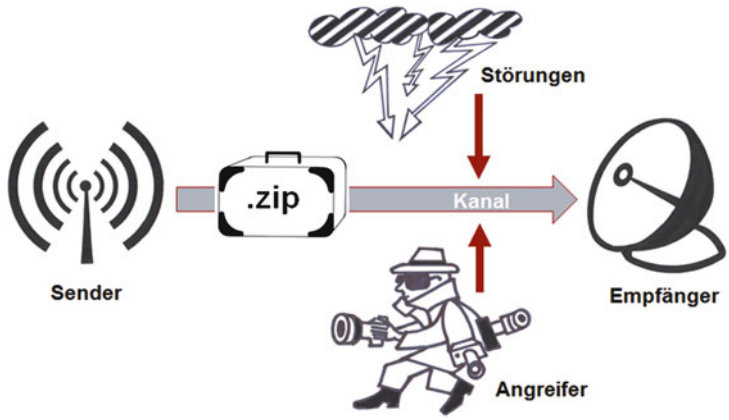


Abb. 1.1 Szenario der Datenübertragung

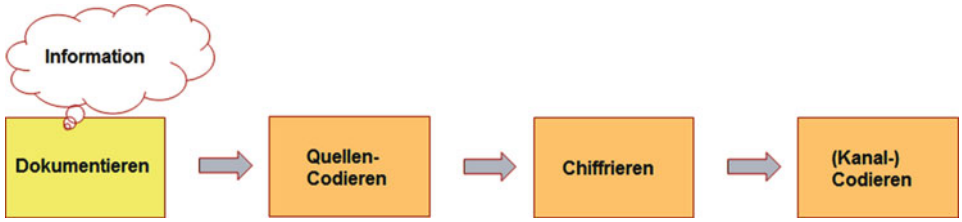


Abb. 1.2 Information versenden

- gegen unerwünschtes Lesen oder gar unerlaubte Veränderung geschützt sein und
- trotz zufälliger Störungen im Übertragungskanal bzw. Beschädigungen verwendeter Speichermedien ohne signifikanten Informationsverlust verfügbar sein.

Je nach Anwendung können die Schwerpunkte dieser Anforderungen unterschiedlich gelagert sein. Umgangssprachlich spricht man dabei häufig in allen drei Fällen von *Codierung*.

1.1.2 Information versenden

Wir wollen daher anhand von Abb. 1.2 zuerst die Begriffe und die einzelnen Schritte etwas genauer klären.

Zunächst muss die gewünschte Information mal *zu Papier* gebracht werden. Das kann etwa in Deutsch, Englisch oder einer anderen Sprache geschehen, und sollte auch mit einigen Grafiken und Fotos illustriert sein.

Als nächstes muss das Dokument in ein *anderes Alphabet* konvertiert werden, in der Regel digitalisiert mit den Bits 0 und 1. Man spricht dabei auch von **Quellencodierung**. Nachrichten sollten dabei so komprimiert werden, dass häufig auftretende Strings in kurzer Form und seltenere in längerer Form codiert sind. Ein klassisches Beispiel ist das Morsealphabet, bei dem für die – im Englischen – häufigsten Buchstaben „e“ und „t“ jeweils „1× kurz“ bzw. „1× lang“ verwendet wird. Modernere Beispiele sind **zip**-komprimierte Datenarchive oder die Dateiformate **jpeg** und **tiff** zur Kompression von Fotos und Grafiken. Wie man einen Text oder allgemeiner eine Information effizient digitalisieren und den Gegebenheiten des Übertragungskanals anpassen kann, werden wir hier nicht behandeln. Dies ist ein Teilaspekt der **Informationstheorie**, die auch **stochastische** Methoden verwendet. Bei der Informationstheorie stehen Begriffe wie *sparsames Codieren*, *Entropie* und *Kanalkapazität* im Vordergrund.

In einem weiteren Schritt sollte unser digitalisiertes Dokument gegen das Abhören oder gar gegen Änderungen durch unbefugte Dritte geschützt werden. Hierzu verschlüsseln wir unseren Text (sog. **Chiffrieren**). Ein berühmtes Beispiel ist die sog. **Cäsar-Chiffre**, bei der jeder Buchstabe im Alphabet durch den drei Stellen weiter stehenden ersetzt wird,

also „a“ durch „d“ und schließlich „z“ durch „c“. Dieses Verfahren ist natürlich leicht zu knacken. Aber auch hier gibt es ausgefeilte Methoden, um den Angreifer auszutricksen, wie z. B. den **AES Advanced Encryption Standard**. Das zugehörige Fachgebiet heißt **Kryptografie**, und soll hier auch nicht weiter vertieft werden. Nur so viel: Man unterscheidet zwischen **symmetrischen Chiffren**, bei denen mit dem Verschlüsselungsalgorithmus auch unmittelbar der zur Entschlüsselung bekannt ist, und **asymmetrischen Chiffren**, bei denen das nicht oder nur extrem schwer und zeitaufwendig möglich ist. Die Cäsar-Chiffre und der AES gehören zu den symmetrischen Verfahren, prominentestes Beispiel der asymmetrischen Chiffren ist das **Public-Key-Verfahren** von Rivest, Shamir und Adleman (**RSA**).

Last but not least ist unser Übertragungskanal störanfällig (z. B. kurzzeitiges Rauschen) oder das verwendete Speichermedium könnte beschädigt worden sein (z. B. Kratzer in der DVD). Bei der sog. **Kanalcodierung** (oder kurz **Codierung**) fügen wir unserem Text redundante Information bei, so dass auftretende Fehler in der Regel erkannt und möglicherweise auch ohne Nachfrage korrigiert werden können. Dieser Schritt ist Inhalt der **Codierungstheorie** und wird im Folgenden ausführlich behandelt.

Als Austauschkanal werden u. a. folgende Medien betrachtet:

- Telefonleitungen,
- Computernetzwerke, z. B. LAN, Internet,
- Fernseh- und Funkübertragungen,
- Satellitenkommunikation,
- Datentransfer zu und von Raumsonden.

Die Speichermedien umfassen z. B.:

- Festplatten,
- USB-Sticks und Speicherchips,
- BAR-Codes,
- CD und DVD.

1.1.3 Information empfangen

Nach dem Empfang unseres Dokuments bzw. beim Auslesen des entsprechenden Speicherinhalts müssen die oben genannten Schritte sämtlich wieder rückgängig gemacht werden, wie es Abb. 1.3 zeigt.

Empfangsfehler sollten zumindest entdeckt, oder noch besser automatisch korrigiert werden. Außerdem muss die Redundanz wieder entfernt und damit die ursprüngliche Nachricht zurückgewonnen werden (**Kanaldecodieren**). Anschließend muss die Nachricht entschlüsselt werden (**Dechiffrieren**), was natürlich voraussetzt, dass der Empfänger den Dechiffrieralgorithmus kennt. Zuletzt muss das Dokument aus seinem digitalisierten

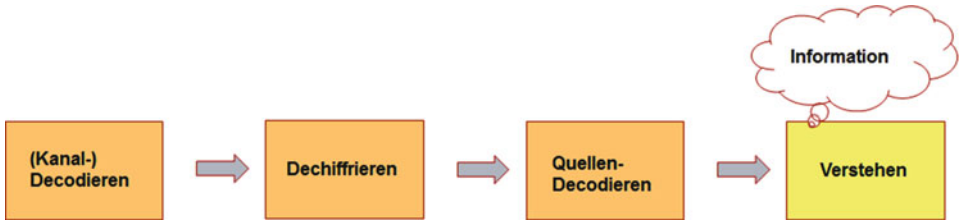


Abb. 1.3 Information empfangen

Zustand **quellendecodiert** und damit wieder in den lesbaren Ausgangstext (einschl. Grafiken) zurück konvertiert werden. Erst jetzt kann der Inhalt des Dokuments vom Empfänger *verstanden* werden.

1.1.4 Inhalt der Codierungstheorie

Hier also nochmals zusammengefasst die Hauptaufgaben der **Codierungstheorie**.

- Konstruktion geeigneter Codes zur Erkennung und Korrektur von möglichst vielen Fehlern unter Verwendung von möglichst wenig redundanter Information,
- Entwicklung effizienter Codier- und Decodieralgorithmen.

Es gibt zwei große Klassen von fehlerkorrigierenden Codes.

- **Blockcodes**,
- **Faltungscodes** (engl. **Convolutional Codes**).

Während Blockcodes auch theoretisch sehr umfänglich untersucht sind, sind die mathematischen Grundlagen von Faltungscodes nicht allzu weit entwickelt. Allerdings erlauben Letztere sehr schnelle und effiziente Codier- und Decodierverfahren, weshalb sie dennoch verbreitet Anwendung finden. Wir konzentrieren uns deshalb zunächst auf Blockcodes, werden aber anschließend auch auf Faltungscodes eingehen.

1.2 Von Parität, Büchern und Euros

Natürliche Sprachen beinhalten schon das Prinzip von Redundanz als Möglichkeit der Fehlererkennung. Wir werden jetzt Beispiele aus der Praxis vorstellen, an denen bereits einige wichtige Prinzipien der Kanalcodierung (im Folgenden stets kurz **Codierung** genannt) sichtbar werden.

1.2.1 Beispiel: Paritätsprüfungscode

Die erste Methode kennen sicher alle. An einen binären Nachrichtenstring mit 0 und 1 (sog. **Bits**) hängt man ein weiteres Bit an und macht damit die Anzahl der Einsen gerade: Bei ungerader Anzahl von Einsen in der Nachricht eine zusätzliche 1, bei gerader Anzahl von Einsen eine 0. Man spricht dann davon, dass die **Parität** des Strings gerade ist. Erhält ein Empfänger einen solchen Nachrichtenstring, so überprüft er die Parität. Ist die Parität des empfangenen Strings ungerade, so weiß der Empfänger, dass bei der Übertragung der Nachricht mindestens ein Fehler aufgetreten ist.

- Weiß er auch wie viele Fehler? Nein, denn es könnte nur ein Bit falsch sein, aber auch drei, fünf, sieben, ..., also jede ungerade Anzahl.
- Was weiß er, wenn die Parität gerade ist? Eigentlich gar nichts. Denn die Nachricht könnte korrekt sein (also null Fehler) oder es könnten auch zwei, vier, sechs, ... Fehler aufgetreten sein, also jede gerade Anzahl.

Wenn wir also wissen, dass die Datenübertragung so gut ist, dass höchstens *ein* Fehler auftreten kann, so wird ein solcher mit dem **Paritätsprüfungscode** sicher erkannt. Werden jedoch bei schlechter Übertragungsqualität üblicherweise mehrere Bits geändert, so ist der Code weitgehend nutzlos. Wir formalisieren unsere Überlegung und nehmen dazu an, dass die Nachrichten Strings der Länge 2 über dem **binären Alphabet {0,1}** sind, also 00, 01, 10 und 11. Als Codierung wählen wir die Paritätsprüfung.

$$00 \to 000, \quad 01 \to 011, \quad 10 \to 101, \quad 11 \to 110.$$

Eine Nachricht x_1x_2 wird also zu $x_1x_2 \to x_1x_2x_1 + x_2$, wobei hier die wohlbekannte Addition von binären Zeichen gemeint ist, also $1+0 = 0+1 = 1$ und $0+0 = 1+1 = 0$. Dies ist also ein kleiner Paritätsprüfungscode mit Codewörtern der Länge 3.

1.2.2 Anwendung: ASCII-Zeichen

Die 128 sog. **ASCII-Zeichen** werden auch mittels Paritätsprüfung codiert. Für die 128 Zeichen benötigt man nur sieben Bits (das ergibt $2^7 = 128$ Kombinationen), man verwendet aber acht Bits, wobei das achte Bit ein Paritätsbit ist, d. h. $x_8 = x_1 + \ldots + x_7$. Heute werden jedoch meist die $2^8 = 256$ Zeichen für den sog. **erweiterten ASCII-Zeichensatz** verwendet, also ohne Paritätsprüfung.

1.2.3 Beispiel: Wiederholungscode

Mit dem Paritätsprüfungscode kann man zwar einen Fehler erkennen, aber man weiß natürlich nicht, an welcher Stelle der Bitfolge der Fehler aufgetreten ist, um den Fehler auch

gleich wieder ohne Nachfrage korrigieren zu können. Wie wäre es also damit: Wir senden die Nachricht als **Wiederholungscode** einfach ein zweites Mal, oder sogar dreimal. Dann sollten doch zwei der drei identischen Nachrichtenstrings auch nach Datenübertragung identisch sein und wir kennen dann sofort die richtige Nachricht ohne weitere Nachfrage. Wir wollen auch das etwas formalisieren. Nachrichten und das Alphabet seien dieselben wie bei der Paritätsprüfung, jedoch machen wir den Code jetzt sehr redundant, wir senden nämlich die Nachricht einfach dreimal, und bilden daher Codewörter der Länge 6.

$$\begin{aligned} 00 &\to 000000, \\ 01 &\to 010101, \\ 10 &\to 101010, \\ 11 &\to 111111. \end{aligned}$$

Zwei verschiedene Codewörter unterscheiden sich offenbar an mindestens drei Stellen. Tritt also bei der Übertragung eines Codeworts nur ein Fehler auf, so kann er automatisch korrigiert werden, indem man einfach das *eine* Codewort wählt, das sich nur an einer Stelle vom empfangenen Wort unterscheidet. Beim Auftreten von bis zu zwei Fehlern wird zumindest erkannt, dass das empfangene Wort fehlerhaft ist, die Korrektur nach obiger Methode liefert aber leider nicht mehr das korrekte Ergebnis. Die bessere Eigenschaft bei der Fehlererkennung und -korrektur gegenüber dem Paritätsprüfungscode haben wir aber leider mit vierfach größerer Redundanz und damit geringerer Übertragungsrate erkauft.

1.2.4 Anwendung: Die alte ISBN-Buchnummer

Die bis 2006 gebräuchliche **ISBN** (**International Standard Book Number**) ist ein zehnstelliger Code, der jedes Buch international identifizierbar macht.

Stelle 1	Sprachregion des Verlagssitzes (z. B. 0 für Englisch, 3 für Deutsch),
Stellen 2–4	Verlagsnummer,
Stellen 5–9	individuelle Buchnummer,
Stelle 10	Prüfziffer.

Die ersten neun Stellen verwenden als Alphabet die Ziffern $\{0, 1, \ldots, 9\}$, für die Prüfziffer dagegen wird $\{0, 1, \ldots, 9, \mathrm{X}\}$ genutzt, wobei X für die Zahl 10 steht. Der *Klassiker* von Jacobus van Lint „Introduction to Coding Theory“ [vLi82] hat z. B. die ISBNs 0-387-11284-7 und 3-540-11284-7. Die Prüfziffer c_{10} eines ISBN-Codewortes $c_1 \ldots c_{10}$ berechnet sich aus

$$10c_1 + 9c_2 + \ldots + 2c_9 + c_{10} = \sum_i (11 - i)c_i = 0 \pmod{11}.$$

Hier wird also **modulo** 11 (kurz **mod** 11) gerechnet, d. h. mit Resten bei Division durch die Zahl 11. Die Aussage „$= 0$“ kann man also auch als „Teilbarkeit durch 11“ lesen. Man beachte, dass die Addition binärer Zahlen nichts anderes ist als die Addition modulo 2 von natürlichen Zahlen. Macht man beim Abschreiben einer ISBN einen Fehler, so kann dieser erkannt werden. Wird nämlich an Stelle i statt c_i der Wert x geschrieben, und erfüllt auch die Ziffernfolge mit x an der Stelle i die obige Prüfbedingung, dann ergibt sich $(11-i)c_i = (11-i)x \pmod{11}$, d. h. $(11-i)(c_i - x) = 0 \pmod{11}$. Da 11 eine Primzahl ist und $0 < i < 11$ gilt, muss 11 die Differenz $c_i - x$ teilen, also $c_i = x$.

Außerdem erkennt man, wenn zwei beliebige Stellen versehentlich miteinander vertauscht wurden. Nehmen wir dazu an, c_i und c_j (für $j > i$) wurden vertauscht und auch das Wort mit c_j an Stelle i und c_i an Stelle j ist wieder ein ISBN-Codewort. Dann gilt

$$(11-i)c_i + (11-j)c_j = (11-i)c_j + (11-j)c_i \pmod{11}$$

und folglich $(j-i)(c_i - c_j) = 0 \pmod{11}$. Da aber 11 eine Primzahl ist, muss 11 entweder $j - i$ oder $c_i - c_j$ teilen. Wegen $1 < j - i < 9$ folgt $c_i = c_j$.

Wie die neue ISBN-Nummer aussieht, werden wir später noch sehen, wenn wir uns in Abschn. 1.4 um den EAN-Code kümmern.

1.2.5 Anwendung: Die neue Wertpapierkennnummer ISIN

Die **ISIN** (**International Securities Identification Number**) ersetzt seit 2003 die alte deutsche Wertpapierkennnummer (WKN) eines Wertpapiers (Aktien, Fonds, Anleihen etc.). Es handelt sich dabei um eine 12-stellige Buchstaben-Ziffern-Kombination. Die Aktien der Siemens AG haben beispielsweise die ISIN DE0007236101, die der Bayer AG DE000BAY0017. Dabei steht DE für Deutschland, CH für Schweiz usw. (sog. Ländercode), sonst ist der Aufbau länderspezifisch. Die letzte Ziffer aber ist in jedem Fall wieder eine Prüfziffer. Diese berechnet sich folgendermaßen: In der eigentlichen elfstelligen Buchstaben-Ziffern-Kombination werden die Buchstaben durch eine Zahl zwischen 10 und 35 ersetzt, und zwar entsprechend dem Alphabet $A = 10$ bis $Z = 35$, und diese als zwei einstellige Ziffern aufgefasst. Dadurch wird die ursprünglich elfstellige Folge zwar länger, deren einzelne Komponenten bestehen aber nur noch aus einer Ziffer zwischen 0 und 9. Am Beispiel der Siemens-Aktie bedeutet dies 1314000723610. Sei also $x_1 \ldots x_m$ die so gebildete Ziffernfolge und x_{m+1} die zu ermittelnde Prüfziffer, die angehängt werden soll. Sei außerdem $Q()$ die Quersumme einer natürlichen Zahl. Dann wird – beginnend von hinten mit der Prüfziffer – jede zweite Ziffer x_i mit 2 gewichtet und die Prüfziffer x_{m+1} aus

$$x_{m+1} + Q(2x_m) + x_{m-1} + Q(2x_{m-2}) + x_{m-3} + \ldots = 0 \pmod{10}$$

berechnet (sog. **Luhn-Algorithmus**), wobei also hier mit Resten modulo 10 gerechnet wird. Man beachte, dass abhängig davon, ob m gerade oder ungerade ist, der Summand

für x_1 entweder gleich x_1 oder gleich $Q(2x_1)$ ist. Um damit besser rechnen zu können, überlegt man sich vorab, dass für $0 \leq z < 5$ natürlich $Q(2z) = 2z$ gilt, aber für $5 \leq z \leq 9$ die Gleichung $Q(2z) = 1 + (2z - 10) = 2z - 9$ erfüllt ist.

Wir überzeugen uns nun, dass der ISIN-Code einen Ziffernfehler erkennt. Es sei dazu im ISIN-Code die Ziffer x_i in die Ziffer y geändert worden, wobei auch die neue Folge wieder ein gültiges ISIN-Codewort sei. Dann können wir annehmen, dass $Q(2x_i) = Q(2y) \pmod{10}$ gilt. Nun verwendet man die Vorbemerkung und unterscheidet die dortigen Fälle. Wir untersuchen beispielhaft einen, es seien etwa x_i und y größer gleich 5. Dann gilt also $2x_i - 9 = 2y - 9 \pmod{10}$, und damit teilt 10 die Zahl $2(x_i - y)$. Dann teilt aber 5 die Zahl $x_i - y$, was wegen $5 \leq x_i, y \leq 9$ die Gleichheit $x_i = y$ zur Folge hat.

Leider erkennt aber der ISIN-Code nicht jeden Buchstabenfehler. Man muss nämlich berücksichtigen, dass ein solcher sich auf zwei benachbarte Ziffern in der Prüfbedingung auswirkt. Zum Beispiel sind sowohl DE000BAY0017 als auch DE000BAQ0017 gültige ISIN-Codes.

Der ISIN-Code erkennt auch die Vertauschung von zwei benachbarten Ziffern in einem Codewort, sofern diese nicht 0 und 9 sind. Man sollte daher solche Konstellationen vermeiden, was in der Praxis allerdings nicht immer geschieht. Wir wollen uns auch das überlegen und nehmen dazu an, dass die zwei benachbarten Ziffern x_i und x_{i+1} vertauscht wurden und wir trotzdem wieder ein korrektes Codewort erhalten. Dann ist $x_i + Q(2x_{i+1}) = x_{i+1} + Q(2x_i) \pmod{10}$. Sind x_i und x_{i+1} entweder beide kleiner als 5 oder beide größer gleich 5, so ist nach unserer Vorbemerkung $x_i = x_{i+1}$. Sei daher x_i kleiner als 5 und x_{i+1} größer gleich 5. Dann ist also $x_i + 2x_{i+1} - 9 = x_{i+1} + 2x_i \pmod{10}$, folglich $x_{i+1} = x_i + 9 \pmod{10}$ und daher $x_i = 0$ und $x_{i+1} = 9$.

1.2.6 Anwendung: Seriennummern der Euroscheine

Bei **Euroscheinen** der 1. Serie 2002 bestehen die **Seriennummern** aus einem führenden Buchstaben zur Länderkennung (X für Deutschland, S für Italien etc.), einer zehnstelligen, länderspezifisch aufgebauten Ziffernfolge, sowie einer Prüfziffer. Die Gesamtzeichenlänge ist daher 12. Zur Berechnung der Prüfziffer wird zunächst der führende Buchstabe durch seine Position im Alphabet ersetzt, d. h. $A = 01$, $B = 02, \ldots, Z = 26$. Sei also $x_1 \ldots x_{12}x_{13}$ die sich so ergebende Ziffernfolge, dann berechnet sich die Prüfziffer $0 < x_{13} \leq 9$ aus $x_1 + x_2 + \ldots + x_{12} + x_{13} = 8 \pmod{9}$.

Es kann dabei ein Fehler erkannt werden, nicht jedoch der Fehler „$0 \leftrightarrow 9$“ sowie der ein oder andere Buchstabenfehler; die hierfür notwendigen Überlegungen überlassen wir dem Leser als Übung.

Unmittelbar klar ist jedoch, dass eine Vertauschung von Ziffern nicht erkannt werden kann. Warum rechnet man aber modulo 9? Der Grund hierfür ist, dass Reste modulo 9 einfach über iterierte Quersummenbildung bestimmt werden können und man auf diese Weise leicht eine Euroscheinnummer auf Gültigkeit überprüfen kann.

In der 2. Serie der Euroscheine ab 2013 wurde das Nummernsystem leicht modifiziert. Die Seriennummern beginnen mit zwei Buchstaben, gefolgt von einer neunstelligen Kennziffer und einer Prüfziffer, d. h. die Seriennummer ist ebenfalls zwölfstellig. Die erste Stelle der Seriennummer gibt dabei die Druckerei an, in der die Banknote hergestellt wurde, während die zweite Stelle sowie die folgenden neun Ziffern zur eindeutigen Kennzeichnung der Banknote innerhalb der Ausgabe einer Druckerei gehören. Ersetzt man die zwei Buchstaben durch ihre Position im Alphabet (d. h. $A = 01, \ldots, Z = 26$), so ist bei Banknoten der zweiten Serie der Neunerrest der Summe über alle 14 Ziffern gleich 7, d. h. der Prüfalgorithmus ist prinzipiell noch derselbe wie bei den Banknoten der ersten Serie.

1.3 Blockcodes – Hamming-Abstand und Fehlerkorrektur

Nachdem wir im letzten Abschnitt das Thema **Codierung** eher intuitiv anhand von einfachen Beispielen angegangen sind, wollen wir uns jetzt der Sache etwas systematischer nähern. Grundsätzlich geht es also darum, Übertragungsfehler, die z. B. durch Rauschen in einem Kanal aufgetreten sind, erkennen und wenn möglich selbstständig korrigieren zu können. Hierzu brauchen wir zunächst einige Grundbegriffe.

1.3.1 Was versteht man unter Blockcodes?

Sei A eine endliche Menge (das **Alphabet**) und n eine natürliche Zahl. Ein **Blockcode** C der **Länge** n über A ist eine nicht-leere Teilmenge aller n-Tupel $A^n = \{(a_1, \ldots, a_n) | a_i \in A\}$. Die Elemente von C heißen **Codewörter**.

Umfasst C genau m Codewörter, und ist B eine Menge von maximal m zu codierenden **Informationseinheiten**, dann ist jede eineindeutige Zuordnung „Element von B“→„Codewort in C“ eine **Codierfunktion**.

Während wir bei unseren bisherigen Beispielen die Codewörter als Strings von Symbolen geschrieben haben, werden wir von nun an die Schreibweise mit n-Tupeln wählen. In der Praxis besteht B meist aus k-Tupeln über dem gleichen Alphabet A. In unseren Eingangsbeispielen haben wir die Zuordnung eines k-Tupels in B zu einem n-Tupel in C durch eine Verlängerung der Zeichensequenz um eine oder mehrere Stellen erreicht. Der Paritätsprüfungscode etwa hat als Alphabet $A = \{0, 1\}$ und ordnet $B = \{(0, 0), (1, 0), (0, 1), (1, 1)\}$ folgende Codewörter in $A^3 = \{0, 1\}^3$ zu.

$$(0, 0) \to (0, 0, 0), (1, 0) \to (1, 0, 1), (0, 1) \to (0, 1, 1), (1, 1) \to (1, 1, 0).$$

1.3.2 Hamming- und Minimalabstand

Entscheidend für die Fehlererkennung und -korrektur ist nun folgende Begriffsbildung.

Sei A ein Alphabet, n eine natürliche Zahl und $a = (a_1, \ldots, a_n), b = (b_1, \ldots, b_n) \in A^n$. Dann heißt die Anzahl der Stellen, an denen sich a und b unterscheiden, der **Hamming-Abstand** von a und b. Man bezeichnet ihn kurz durch

$$d(a, b) = |\{i | 1 \leq i \leq n, a_i \neq b_i\}|.$$

Sei außerdem C ein Blockcode in A^n mit $|C| > 1$. Dann heißt $d(C) = \min\{d(x, y) | x, y \in C, x \neq y\}$ der **Minimalabstand** von C. Für $|C| = 1$ setzt man sinngemäß $d(C) = 0$.

Wird also ein Wort $x \in C$ gesendet und $y \in A^n$ empfangen mit $d(x, y) = d$, so sind genau d Fehler aufgetreten. Das Konzept des Hamming-Abstands stammt aus dem Jahr 1950 von **Richard W. Hamming** (1915–1998), einem amerikanischer Mathematiker, dessen Arbeit großen Einfluss auf die Informatik und Telekommunikation hatte. Er gilt als Mitbegründer der Codierungstheorie.

Wir betrachten ein Beispiel, nämlich $A = \{0, 1\}$ und die 3-Tupel A^3. Man kann sich diese wie einen Würfel vorstellen. Der Hamming-Abstand ist dann die Anzahl der Kanten zwischen zwei 3-Tupeln, wie Abb. 1.4 zeigt. Für größere A^n wird diese räumliche Vorstellung natürlich schwieriger.

1.3.3 Eigenschaften des Hamming-Abstands

Jedenfalls hat dieser Abstandsbegriff viele *vernünftige* Eigenschaften, die alle fast unmittelbar klar sind. Sei A ein Alphabet und $a, b, c \in A^n$. Dann gilt.

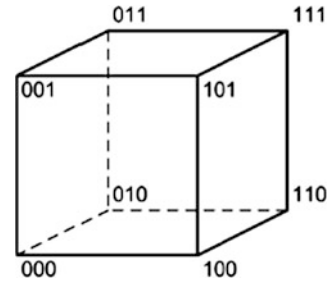


Abb. 1.4 Visualisierung des Hamming-Abstands

- $0 \leq d(a,b) \leq n$;
- $d(a,b) = 0$ genau dann wenn $a = b$;
- $d(a,b) = d(b,a)$;
- $d(a,c) \leq d(a,b) + d(b,c)$, sog. **Dreiecksungleichung**.

Zum Nachweis des vierten Punkts überlegt man sich am besten zuerst: Unterscheiden sich a und c an einer Stelle i, so unterscheiden sich entweder a und b an der Stelle i oder b und c.

1.3.4 Geometrische Interpretation des Hamming-Abstands

Wie in einem euklidischen Raum können wir jetzt **Kugeln** von einem Radius r um den **Mittelpunkt** $a \in A^n$ bilden, also $B_r(a) = \{x \in A^n | d(a,x) \leq r\}$.

Aber wie viele Elemente $x \in A^n$ enthält denn eine solche Kugel $B_r(a)$? Sei dazu $|A| = q$. In $B_r(a)$ gibt es zunächst einmal a selbst. Die Anzahl der n-Tupel x, die sich an genau einer Stelle von a unterscheiden, ist $n(q-1)$; die Anzahl der n-Tupel x, die sich an genau zwei Stellen von a unterscheiden, ist $\binom{n}{2}(q-1)^2$. Allgemein gibt es folglich $\binom{n}{i}(q-1)^i$ n-Tupel, die sich an genau i Stellen von a unterscheiden. Zur Erinnerung: $\binom{n}{i}$ ist der **Binomialkoeffizient** und gibt die Anzahl der i-elementigen Teilmengen einer n-elementigen Menge an. Dabei gilt $\binom{n}{i} = n!/(i!(n-i)!)$, wobei n-**Fakultät** $n!$ das Produkt aller natürlichen Zahlen kleiner oder gleich n ist.

Wir haben uns also überlegt, dass $|B_r(a)| = \sum_{i=0}^{r} \binom{n}{i}(q-1)^i$ gilt.

Diese Kugeln können wir auch aufmalen, obwohl wir uns das Ganze dann wieder in einer ganz normalen euklidischen Ebene vorstellen. Aber die Anschauung hilft bei folgender Erkenntnis.

1.3.5 Fehlererkennung und -korrektur

Ein Blockcode C heißt ***e*-fehlerkorrigierend**, falls die Kugeln $B_e(c)$ und $B_e(c')$ um je zwei verschieden Codewörter $c, c' \in C$ paarweise disjunkt sind, d. h. für den Minimalabstand d von C die Beziehung $d \geq 2e+1$ gilt. Ist e maximal gewählt, so spricht man auch von der **Fehlerkorrekturkapazität** von C. Die Situation ist in Abb. 1.5 veranschaulicht. Wenn nämlich bei der Übertragung eines Codeworts c höchstens e Fehler auftauchen, so liegt das empfangene n-Tupel $v \in A^n$ nur in der *einen* Kugel um c und in keiner anderen und wir können eindeutig nach c korrigieren.

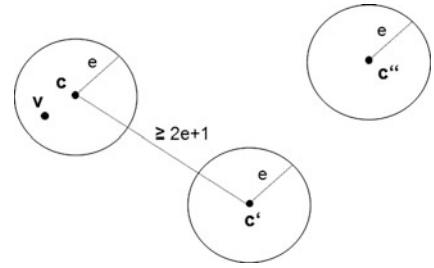


Abb. 1.5 Disjunkte Hamming-Kugeln

Bei den meisten technisch implementierten Verfahren wird die empfangene Information automatisch korrigiert (sog. **FEC**, **Forward Error Correction**). Allerdings kann man sich nie ganz sicher sein, dass bei der Übertragung eines Codeworts wirklich nur höchstens e Fehler aufgetreten sind.

Ein Blockcode C heißt ***t*-fehlererkennend**, falls für alle Codewörter $c \in C$ die Kugel $B_t(c)$ außer c kein weiteres Codewort enthält, d. h. für den Minimalabstand d von C die Beziehung $d \geq t + 1$ gilt. Wenn also bei der Übertragung eines Codeworts c höchstens t Fehler auftreten, so kann das empfangene n-Tupel $v \in A^n$ kein anderes Codewort sein, und wir können erkennen, dass Fehler aufgetreten sind. Es könnte aber passieren, dass v im Sinne des Hamming-Abstands *näher* an einem anderen Codewort c' liegt. Eine eindeutige Korrektur ist daher nicht möglich.

Dieses Verfahren wird dann eingesetzt, wenn z. B. ein Computernetzwerk nach Fehlererkennung eine Wiederholung der Übertragung beim Sender anfordert (sog. **ARQ**, **Automatic Repeat reQuest**).

1.3.6 Auslöschungen

In den meisten digitalen Übertragungskanälen werden Bits schlimmstenfalls verfälscht, also 0 zu 1 oder 1 zu 0, aber sie gehen in der Regel nicht *verloren*. Dennoch gibt es auch solche Fälle, das Standardbeispiel sind Kratzer auf CDs. Man spricht dabei formal von Auslöschungen und meint damit, dass bei einem gesendeten oder gespeicherten Codewort $c = (c_1, \ldots, c_n)$ einige Stellen nicht empfangen bzw. gelesen werden können. Wir nehmen der Einfachheit halber an, es handele sich um die ersten a Stellen und schreiben das empfangene bzw. gelesene Wort v als $v = (*, \ldots, *, c_{a+1}, \ldots, c_n)$.

Zunächst mag man denken, dass das noch viel schlimmer ist als a Fehler. Aber im Gegensatz zu Fehlern kennt man bei Auslöschungen ja deren Anzahl und Positionen. Ist nun $d \geq a+1$ für den Minimalabstand d von C, und korrigieren wir v zu einem Codewort $c' = (c'_1, \ldots, c'_a, c_{a+1}, \ldots, c_n)$, so gilt $d(c, c') \leq a \leq d-1$ und folglich $c' = c$. Ein Code

ist also ***a*-auslöschungskorrigierend**, wenn $d \geq a + 1$ gilt. Die Fehlerkorrekturkapazität e ist also kleiner als die für Auslöschungen.

1.3.7 BD-Decodierung

Sei e die Fehlerkorrekturkapazität des Codes C. Auch wenn es wünschenswert wäre, dass bei der Übertragung eines Codeworts höchstens e Fehler auftreten, so können wir leider nicht davon ausgehen, dass das auch immer so ist. Wir sehen es dem empfangenen Wort eben nicht an. Also muss man sich eine gängige Lösung überlegen, wie man in der Praxis damit umgeht.

- Eine Möglichkeit ist, das empfangene Wort v stets zu einem Codewort c mit minimalem Hamming-Abstand zu korrigieren. Dann hat man, auch wenn der Abstand zwischen v und c größer als e ist, zumindest eine Chance, dass man richtig liegt.
- Eine zweite Möglichkeit ist, dass man im Fall eines *großen* Abstands zwischen empfangenem Wort v und dem nächsten Codewort (z. B. größer als e) nicht korrigiert, und v ggf. entsprechend markiert (etwa mit „?“). Diese Art der Decodierung nennt man **BD-Decodierung** (**Bounded Distance Decoding**). Der Vorteil von BD-Decodierung ist einerseits, dass man möglicherweise sinnlose Decodierungen vermeidet. Andererseits sind viele schnelle Decodierverfahren von Hause aus BD-Verfahren.

1.3.8 Beispiel: Fehlererkennung und -korrektur

Wir schauen uns vor diesem Hintergrund nochmals einige Beispiele aus dem letzten Abschnitt an.

Paritätsprüfungscode
Der Paritätsprüfungscode ist 1-fehlererkennend, wie wir im letzten Abschnitt gesehen haben.

Kann man einen Fehler aber auch korrigieren, d. h., ist der Code auch 1-fehlerkorrigierend? Nein, das haben wir uns auch schon klar gemacht. Überlegen wir uns das aber noch mal formaler mit dem Hamming-Abstand: Wird z. B. $(0, 0, 1)$ empfangen, so ist wegen der Parität klar, dass ein Fehler aufgetreten sein muss. Aber der String könnte sowohl aus dem Codewort $(0, 0, 0)$ als auch $(1, 0, 1)$ entstanden sein, beide haben nämlich Hamming-Abstand 1 zu $(0, 0, 1)$. Dies passt auch damit zusammen, dass für den Minimalabstand d des Codes offenbar $d = 2$ gilt.

Wiederholungscode
Der Wiederholungscode muss nach unseren Überlegungen 1-fehlerkorrigierend bzw. 2-fehlererkennend sein, da sein Minimalabstand $d = 3$ ist. Aus dieser Sicht ist er also etwas besser als der Paritätsprüfungscode, aber auch viel redundanter.

ISBN-Code
Unterscheiden sich beim ISBN-Code zwei Codewörter c und c' nur einmal bei den ersten 9 Stellen, sagen wir an der Stelle i, so sind in jedem Fall auch ihre jeweiligen Prüfziffern an Stelle 10 verschieden. Sonst wäre nämlich $(11-i)c_i = (11-i)c_i' \pmod{11}$ und folglich $i(c_i - c_i') = 0 \pmod{11}$. Da 11 kein Teiler von i ist, müsste doch $c_i = c_i'$ gelten. Daher ist der Minimalabstand $d \geq 2$. Wenn sich andererseits zwei ISBN bei den eigentlichen Nummern nur an der letzten Stelle der individuellen Buchnummer unterscheiden, so unterscheiden sie sich insgesamt dort und bei der Prüfziffer. Also ist auch $d \leq 2$ und wir haben $d = 2$ gezeigt. Der Code ist also 1-fehlererkennend und 0-fehlerkorrigierend.

ISIN-Code
Der ISIN-Code ist – zumindest was Buchstaben betrifft – nicht 1-fehlererkennend. Auch die Seriennummern der Euroscheine sind nicht 1-fehlererkennend, da der Fehler „$0 \leftrightarrow 9$“ nicht erkannt wird.

1.3.9 Äquivalente Codes

Es sollte aus der Definition des Hamming-Abstands klar geworden sein, dass es bei der Güte eines Blockcodes

- nicht auf die Reihenfolge der Tupel ankommt und
- dass auch die Alphabetselemente bei den einzelnen Positionen beliebig permutiert sein können.

Dabei bleiben nämlich die Hamming-Abstände unverändert. Zum Beispiel hätte man die Prüfziffer beim ISBN-Code genauso gut an die erste Stelle setzen können.

Blockcodes, die sich nur dadurch unterscheiden, dass die Reihenfolge der Tupel verändert ist und an den einzelnen Positionen die Alphabetselemente permutiert sind, nennt man daher **äquivalent**.

1.3.10 Beispiel: Ein kombinatorisch definierter binärer Code

Zum Abschluss dieses Abschnitts noch ein kleiner, auf Basis von Abb. 1.6 kombinatorisch definierter Code. Für vier Datenbits $d_1, \ldots, d_4$ definiert man drei Paritätbits $p_1, \ldots, p_3$. Diese konstruiert man so, indem man zu jedem Tupel $(d_1, \ldots, d_4)$ die entsprechenden Paritätbits im Kreisdiagramm so setzt, dass die Parität in jedem Kreis gerade ist (d. h. die

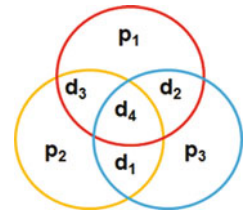


Abb. 1.6 Ein kleiner kombinatorischer Code

bitweise Addition modulo 2 gleich 0 ist). Zum Beispiel setzt man für $d_1 = d_2 = 1$ und $d_3 = d_4 = 0$ die Paritätsbits $p_1 = p_2 = 1$ und $p_3 = 0$.

Hier sind sämtliche Codewörter $(d_1, d_2, d_3, d_4, p_1, p_2, p_3)$ des Codes.

$$\begin{aligned} C = \{&(0,0,0,0,0,0,0), (1,0,0,0,0,1,1), (0,1,0,0,1,0,1), (0,0,1,0,1,1,0),\\ &(0,0,0,1,1,1,1), (1,0,0,1,1,0,0), (0,1,0,1,0,1,0), (0,0,1,1,0,0,1),\\ &(1,1,0,0,1,1,0), (1,0,1,0,1,0,1), (0,1,1,0,0,1,1), (1,1,0,1,0,0,1),\\ &(1,0,1,1,0,1,0), (0,1,1,1,1,0,0), (1,1,1,0,0,0,0), (1,1,1,1,1,1,1)\}. \end{aligned}$$

Wieviel Fehler kann man damit erkennen? Kann man auch Fehler korrigieren? Wie groß ist der Minimalabstand? In Abschn. 2.2 werden wir das auflösen, aber mit einem systematischeren Ansatz. Es handelt sich dabei um einen der berühmtesten Codes in der Geschichte der Codierungstheorie.

1.4 Von Shannon bis zu Artikel- und Kontonummern

1.4.1 Kanalcodierungssatz von Shannon

Im letzten Abschnitt haben wir die wichtigsten Begriffe der Codierungstheorie zusammengestellt. Diesen Abschnitt beginnen wir mit der zentralen *Botschaft*. Wir haben nämlich bisher den Eindruck, dass wir bessere Fehlererkennungs- oder -korrektureigenschaften durch Vergrößerung der Redundanz *erkaufen* müssen. Als Maß für gute Decodiereigenschaft eines Codes haben wir den Minimalabstand kennengelernt. Als Maß für geringe Redundanz benutzt man häufig die sog. **Rate**, welche im Wesentlichen das Verhältnis von Anzahl der Codewörter – genauer gesagt der *Logarithmus* der Anzahl – zur Codelänge ist. Wir kommen darauf noch konkreter zurück. Jetzt aber nur so viel: Je höher die Rate ist, desto geringer ist die Redundanz. Ziel sollte also sein: Man finde Codes mit hoher Rate, aber kleiner Decodierfehlerwahrscheinlichkeit. Der berühmte **Kanalcodierungssatz von Shannon** aus dem Jahr 1948 besagt, dass dieses Ziel prinzipiell beliebig gut erreichbar ist.

Zu jedem vorgegebenen R kleiner als die sog. *Durchsatzkapazität* κ eines Kanals und jedem $\varepsilon > 0$ gibt es einen Code C von möglicherweise sehr großer Länge n,

- dessen Rate einerseits größer als R, aber kleiner als κ ist und
- dessen Decodierfehlerwahrscheinlichkeit kleiner als ε ist.

Claude Shannon (1916–2001) war ein amerikanischer Mathematiker und Elektrotechniker. Er gilt als Begründer der Informationstheorie und Pionier der Codierungstheorie. Seine Beweismethode stammt aus der Informationstheorie. Wir wollen auf den Beweis nicht weiter eingehen, denn es würde uns nicht sonderlich helfen. Es handelt sich nämlich um einen reinen Existenzbeweis und er gibt keinerlei Hilfsmittel, solche Codes auch explizit zu konstruieren. Der Satz von Shannon ist also einerseits Ermutigung, dass man gute Codes finden kann, und somit auch der Startschuss für die Codierungstheorie. Das *Wie* bleibt allerdings offen und ließ bzw. lässt daher noch viel Raum für Erfindergeist, wie wir im Folgenden sehen werden.

1.4.2 Optimale Codes und Gilbert-Schranke

Wir wagen zunächst mal einen Versuch mit der *Brechstange*, gute Codes zu konstruieren.

Wir wollen uns nämlich zu vorgegebenem Alphabet A mit $|A| = q$, sowie vorgegebener Länge n und vorgegebenem Minimalabstand d einen Code C mit möglichst großer Rate vorstellen, also mit $|C|$ möglichst groß. Man nennt solche Codes auch **optimal**, bezogen auf die vorgegebenen Werte q, n und d. Jedes $x \in A^n$ muss dann aber in einer Kugel $B_{d-1}(c)$ für ein geeignetes Codewort $c \in C$ liegen, denn sonst hätte der Code $C \cup \{x\}$ ebenfalls Minimalabstand d und ein Element mehr als C. Also ist $A^n \subseteq \bigcup_c B_{d-1}(c)$, wobei die Vereinigung über alle $c \in C$ gebildet wird. Wir wissen auch bereits

$$|B_{d-1}(c)| = \sum_{i=0}^{d-1} \binom{n}{i} (q-1)^i \text{ und schließen daraus } q^n = |A^n| \leq |C| \sum_{i=0}^{d-1} \binom{n}{i} (q-1)^i.$$

Aus unseren Überlegungen können wir also zwei Folgerungen ziehen. Für **optimale** Codes C gilt einerseits die sog. **Gilbert-Schranke** (1952)

$$|C| \geq q^n / \sum_{i=0}^{d-1} \binom{n}{i} (q-1)^i,$$

die nach dem amerikanischen Mathematiker und Informatiker **Edgar Nelson Gilbert** (1923–2013) benannt ist. Andererseits kann man *gute* Codes mit Gilbert-Schranke so kon-

struieren, indem man mit irgendeinem Wort $c_0 \in A^n$ startet und dann sukzessive weitere Wörter hinzufügt, die Abstand mindestens d von allen vorher gewählten Wörtern besitzen, bis dies nicht mehr möglich ist.

Nehmen wir als Beispiel $A = \{0, 1\}$ und $n = 8$. Für Minimalabstand $d = 3$ hat also ein optimaler binärer Code C der Länge 8 wegen $|C| \geq 2^8/(1+8+28) = 256/37 \sim 6{,}9$ gerade mal mindestens sieben Elemente.

Unser Konstruktionsprinzip funktioniert zwar immer, liefert aber nicht gerade handhabbare Codes. Wir werden in den nächsten Abschnitten sehen, wie man das besser machen kann. Bei all der Suche nach guten Codes mit großem Minimalabstand bei möglichst wenig Redundanz (d. h. großer Rate) darf man aber nie die Praxis aus den Augen lassen. Der Code sollte natürlich auch immer der jeweiligen Anwendung entsprechen.

- Er sollte die Fehlerwahrscheinlichkeit und -art im Übertragungskanal berücksichtigen. Treten in der Regel eher wenige Fehler auf, so muss der Minimalabstand nicht unbedingt künstlich groß gewählt werden. So hat der ISBN-Code nur Minimalabstand 2 und kann daher einen Fehler erkennen, er kann aber zusätzlich Ziffernvertauschungen feststellen.
- Außerdem sollte der Codier- und Decodieralgorithmus der Situation angepasst sein, z. B. bei weniger komplexen Anwendungen auch einfacher berechen- und handhabbar sein.

Mit diesen Vorbemerkungen wenden wir uns nochmals zwei der wichtigsten Praxisanwendungen zu, bevor wir in den nächsten Abschnitten die Sache wieder konzeptioneller angehen werden.

1.4.3 Anwendung: Die Artikelnummer EAN

Die **Europäische Artikelnummer EAN** (auch **GTIN**, **Global Trade Item Number** genannt) ist ein 13-stelliger Code, der sich als Ziffernfolge und als Barcode zum Einscannen auf vielen Produktverpackungen findet. Das Alphabet des Codes ist $\{0, 1, \ldots, 9\}$. Die ersten drei Ziffern beinhalten als Länderpräfix den Sitz des Unternehmens (400–440 steht für Deutschland), die vierte bis zwölfte Ziffer umfassen die Unternehmensnummer und die individuelle Artikelnummer des Herstellers. Die letzte Ziffer ist wieder eine Prüfziffer, die sich für ein Codewort $(c_1, \ldots, c_{13})$ aus der Gleichung

$$c_1 + 3c_2 + c_3 + \ldots + 3c_{12} + c_{13} = 0 \pmod{10}$$

berechnet.

Mit dem EAN-Code kann ein Fehler erkannt werden. Dies überlegt man sich wie bei der ISBN und verwendet dabei, dass 3 und 10 teilerfremd sind. Somit ist der EAN-Code 1-fehlererkennend bei nur einer zusätzlichen redundanten Position. Insbesondere ist der

Abb. 1.7 Die neue ISBN (Buch von Wolfgang Willems, Codierungstheorie und Kryptographie [Wil2])

Abb. 1.8 Der EAN-Code als Barcode

Minimalabstand $d \geq 2$. Wenn sich zwei EANs bei den eigentlichen Nummern nur an Position 12 unterscheiden, dann unterscheiden sich die beiden EANs insgesamt nur an den Positionen 12 und 13. Also ist $d = 2$ und der EAN-Code ist 0-fehlerkorrigierend.

Er kann aber auch Vertauschungen von c_i und c_j erkennen, sofern die Differenz der Indizes ungerade ist und $|c_i - c_j| \neq 5$. Andernfalls wäre nämlich $c_i + 3c_j = c_j + 3c_i \pmod{10}$, und 10 müsste $2(c_i - c_j)$ teilen. Damit wäre aber 5 ein Teiler von $c_i - c_j$, was jedoch ausgeschlossen war.

Im Jahr 2007 wurde die alte ISBN umgestellt, sodass sie nun mit dem EAN-Code kompatibel ist. Dazu wurde jeder ISBN ein Präfix **Buchland** 978 oder 979 vorangestellt, und die Prüfziffer gemäß EAN berechnet. Die Abb. 1.7 zeigt ein Beispiel.

Wir wollen uns jetzt auch noch anhand von Abb. 1.8 die technische Umsetzung des **EAN-Codes** als **Barcode** anschauen. Der Begriff Barcode wird also hier im Sinne einer Quellencodierung verwendet.

Der gesamte Code besteht aus 95 gleich breiten, nebeneinander angeordneten Strichen.

- Strich schwarz $= 1$,
- Strich weiß $= 0$ (auch Freiraum genannt).

Diese 95 Striche gliedern sich von links nach rechts wie folgt:

- 101 als Randzeichen (Beginn des Codes),
- 6 mal 7 Striche, die jeweils eine Ziffer zwischen 0 und 9 codieren,
- 01010 als Trennzeichen (Mitte des Codes),
- 6 mal 7 Striche, die jeweils eine Ziffer zwischen 0 und 9 codieren,
- 101 als Randzeichen (Ende des Codes).

Die Codierung der Ziffern ist sehr ausgeklügelt gewählt. Zum Beispiel gibt es bei jeder Ziffer je zwei (möglicherweise dicke) Linien und zwei (möglicherweise dicke) Freiräume.

Tab. 1.1 Codierungsschema für die BAR-Code-Darstellung des EAN-Codes (Quelle [WPEAN])

Ziffer	Code 1. Ziffer	Linker Block ungerade (U)	Linker Block gerade (G)	Rechter Block
0	UUUUUU	0001101	0100111	1110010
1	UUGUGG	0011001	0110011	1100110
2	UUGGUG	0010011	0011011	1101100
3	UUGGGU	0111101	0100001	1000010
4	UGUUGG	0100011	0011101	1011100
5	UGGUUG	0110001	0111001	1001110
6	UGGGUU	0101111	0000101	1010000
7	UGUGUG	0111011	0010001	1000100
8	UGUGGU	0110111	0001001	1001000
9	UGGUGU	0001011	0010111	1110100

Außerdem sind die Ziffern der linken Hälfte und die der rechten Hälfte unterschiedlich codiert. In der linken Hälfte gibt es zusätzlich für ein- und dieselbe Ziffer sogar zwei Codierungen (gerade und ungerade genannt), die eine weitere Information enthalten. Wer mitgezählt hat, stellt nämlich fest, dass die obige Beschreibung bis jetzt nur 12 Ziffern enthält, wir brauchen für EAN aber 13. Die Konstellation, mit der in der linken Hälfte die Ziffern als gerade (G) oder ungerade (U) codiert sind, legt nämlich die führende Ziffer im EAN-Code fest. Tab. 1.1 enthält alle Details.

Der Aufbau des Barcodes macht es gleichgültig, in welcher Richtung der Artikel über das Lesegerät geführt wird, und ermöglicht eine Schrägabtastung bis zu einem Winkel von etwa 45°.

1.4.4 Anwendung: Die neue Kontonummer IBAN

Die **IBAN** (**International Bank Account Number**) ist eine internationale, standardisierte Notation für Bankkontonummern. Die IBAN wurde entwickelt, um die Zahlungsverkehrssysteme der einzelnen Länder einheitlicher zu gestalten. Sie ist wie folgt strukturiert.

- zweistelliger Ländercode (bestehend aus Buchstaben),
- zweistellige Prüfziffer (bestehend aus Ziffern),
- max. 30-stellige Kontoidentifikation (bestehend aus Buchstaben oder Ziffern).

Die Struktur der Kontoidentifikation (d. h. Länge und Buchstaben-/Ziffernraster) ist durch den Ländercode eindeutig vorgegeben. In Deutschland wird „DE“ als Ländercode genutzt und die Kontoidentifikation besteht aus 18 Ziffern (acht Ziffern für die alte Bankleitzahl und zehn Ziffern für die individuelle Kontonummer). Die Prüfziffer berechnet sich wie folgt.

- Schreibe an die Stelle der zu berechnenden Prüfziffern 00.
- Schiebe die vier ersten Stellen an das Ende der IBAN (z. B. DE00 bei Deutschland).
- Ersetze die Buchstaben im String durch Zahlen, gemäß der Regel $A = 10$, $B = 11$, $\ldots$, $Z = 35$. Dabei wird der String verlängert, da jeder Buchstabe durch zwei Ziffern ersetzt wird.
- Interpretiere den String als natürliche Zahl, ignoriere dabei insbesondere führende Nullen.
- Berechne den Rest dieser natürlichen Zahl bei Division durch 97, d. h. rechne modulo 97.
- Subtrahiere den Rest von 98; falls sich eine ein-stellige Zahl ergibt, mache sie zweistellig mit einer führenden 0.
- Die so berechnete Zahl ist die zweistellige Prüfziffer, die in der IBAN an den Positionen 3 und 4 steht.

Schauen wir uns den Algorithmus genauer an. Sei dazu n die natürliche Zahl, die sich beim vierten Schritt ergibt, ohne Berücksichtigung der beiden letzten Nullen. Dann ist $100n$ also die gesamte natürliche Zahl beim vierten Schritt. Im fünften Schritt dividiert man $100n$ durch 97, also $100n = 97q + r$ mit einem Rest $0 \leq r < 97$. Bei der Überprüfung der IBAN werden die Schritte zwei bis fünf ebenso durchgeführt, wobei jetzt anstelle des Dummys 00 die richtige Prüfziffer $98 - r$ nach hinten gestellt wird. Daher lautet die Zahl jetzt $100n + (98 - r) = (97q + r) + (97 + 1 - r) = 97(q + 1) + 1$. Ist also bei der Überprüfung der IBAN der Rest bei Division durch 97 gleich 1, so lautet das Urteil *o.k.*

Der IBAN-Code erkennt einen Fehler. Um das einzusehen, muss man zunächst berücksichtigen, dass es sich bei einem Fehler wegen der Konvertierung der Buchstaben um bis zu zwei benachbarte falsche Stellen bei der in der Prüfroutine verwendeten natürlichen Zahl handeln kann. Es mögen sich also die beiden Zahlen (d. h. die korrekte und die inkorrekte) um $k_1 10^l + k_2 10^{l+1} = 10^l(k_1 + 10k_2)$ unterscheiden. Dann muss aber 97 die Differenz der beiden Zahlen und damit auch $k_1 + 10k_2$ teilen. Das jedoch geht nur, wenn $k_1 = 7$ und $k_2 = 9$ (oder -7 und -9) ist. Dies ist aber weder ein einstelliger Ziffernfehler noch ein zweistelliger Buchstabenfehler. Somit ist der IBAN-Code 1-fehlererkennend bei allerdings zwei zusätzlichen redundanten Positionen.

Lineare Codes

2

Etwas Algebra – endliche Körper Eines haben wir aus unseren Beispielen gelernt: Wenn wir eine Chance haben wollen, gute Codes zu konstruieren, dann sollten wir mit den Codewörtern in geeigneter Weise *rechnen* können. Wenn wir uns also unsere Codewörter als n-Tupel über einem endlichen Alphabet A vorstellen, so kommt uns doch gleich der euklidische dreidimensionale Raum $\mathbb{R}^3$ in den Sinn oder besser allgemeiner $\mathbb{R}^n$. Mit diesen können wir *rechnen*, nämlich Vektoren komponentenweise addieren und Unterräume davon bilden (z. B. Geraden und Ebenen). Unterräume besitzen **Basisvektoren** und eine **Dimension**. Was hindert uns nun daran, statt des unendlichen **Körpers** $\mathbb{R}$ der reellen Zahlen endliche Körper K zu betrachten, wie z. B. $K = \{0, 1\}$ oder $K = \{0, 1, -1\}$? Eigentlich gar nichts, also tun wir's. Und wenn wir schon dabei sind: Das übliche **Skalarprodukt** von Vektoren in $\mathbb{R}^3$ oder $\mathbb{R}^n$ lässt sich natürlich genauso für Vektoren in K^n bilden.

Grundlagen linearer Codes Mit diesem algebraischen Rüstzeug im *Handgepäck* gehen wir wieder auf die Suche nach guten Codes. Jetzt haben wir aber einen strategischen Vorteil: Wir können uns das Alphabet A als endlichen Körper K vorstellen und gehen kurzerhand dazu über, nur noch **lineare Codes** zu betrachten, d. h. Unterräume des Raumes der n-Tupel K^n über K. Ein linearer Code hat dann eine **Dimension** k und auch **Basisvektoren**, die linear unabhängig sind und den Code aufspannen. Damit haben wir endlich auch ein einfaches Maß für die Redundanz, nämlich die **Rate** k/n: Je größer die Rate, desto kleiner die Redundanz. Außerdem müssen wir uns jetzt nicht mehr alle Codewörter merken, sondern es reicht, einen Satz von Basisvektoren zu kennen. Diese schreiben wir übersichtshalber als Zeilen einer Matrix (mit k Zeilen und n Spalten) und nennen sie **Generatormatrix** unseres Codes. Wir überprüfen unser Konzept an den Eingangsbeispielen sowie an zwei berühmten Codes – dem **kleinen binären Hamming-Code** und dem **ternären Golay-Code**.

O. Manz, *Fehlerkorrigierende Codes*, DOI 10.1007/978-3-658-14652-8_2

Erweiterte und duale Codes Die Idee, mit linearem Code zu arbeiten, scheint ganz brauchbar zu sein. Dann verfolgen wir sie noch ein Stück weiter. Wir können z. B. einen linearen Code um eine Stelle verlängern, indem wir – analog zu unserem ersten Beispiel – dort eine Paritätsprüfung durchführen. Man nennt diesen den **erweiterten Code**. Außerdem bringen wir jetzt das Skalarprodukt *in Stellung* und betrachten die Vektoren, die auf allen unseren Codewörtern *senkrecht* stehen, d. h. Skalarprodukt 0 mit ihnen haben. Man nennt diesen Code den **dualen Code** und seine Generatormatrix die **Kontrollmatrix** unseres Ausgangscodes. Auch diese Konzepte testen wir an den Beispielen des letzten Abschnitts.

Maximum-Likelihood- und Syndromdecodierung Jetzt sollten wir unsere Euphorie aber etwas bremsen und unser Vorgehen nochmals hinterfragen: Was sind denn eigentlich unsere Kriterien an gute Codes? Wir sind bislang von großem Minimalabstand und großer Rate ausgegangen. Das ist auch nicht ganz falsch, aber es fehlt ein weiteres, möglicherweise noch wichtigeres Kriterium: Der Code muss nach Datenübertragung korrekt und möglichst effizient decodierbar sein. Decodieren will man offensichtlich ein empfangenes Wort zu *dem* Codewort, für das es am wahrscheinlichsten ist, dass das empfangene Wort aus ihm entstanden ist (sog. **Maximum-Likelihood-Decodierung**). Wir vergewissern uns zunächst, dass dieses Prinzip – zumindest für in der Praxis gängige Kanäle – mit unserer bisherigen Denkweise der Verwendung des kürzesten Hamming-Abstands übereinstimmt (sog. **Hamming-Decodierung**). Außerdem studieren wir das Standarddecodierverfahren für lineare Codes, die **Syndromdecodierung**.

Codierung linearer Codes und Gesamtszenario Nachdem wir uns mit dem immens wichtigen Thema Decodieren von linearen Codes intensiv auseinandergesetzt haben, wollen wir nun noch überlegen, wie man besonders effizient codieren kann. Am besten wäre es doch, wenn man bei jedem Codewort an den ersten k Stellen die eigentliche Information ablesen könnte und die restlichen $n - k$ Stellen für die redundante Information vorbehalten wären. Wir sehen, dass dies immer möglich ist, und nennen diese Form der Codierung **systematisch**. Das ist jetzt Anlass genug, uns nochmals das **Gesamtszenario der Kanalcodierung und -decodierung** klarzumachen – am besten mit dem kleinen binären Hamming-Code und seiner digitalen Schaltlogik.

2.1 Etwas Algebra – endliche Körper

In unserer Definition von Blockcodes haben wir bislang eine beliebige endliche Menge A als Alphabet zugrunde gelegt. In unseren Beispielen allerdings haben wir Alphabete benutzt, mit denen wir rechnen konnten. Konkret waren dies binäre Ziffern sowie die Reste ganzer Zahlen modulo 97, 11, 10 oder 9. Zu diesem Zweck mussten wir allerdings zusätzlich einige Buchstaben in Ziffern konvertieren. Es wird sich bei unserer Suche nach guten

Codes herausstellen, dass es sinnvoll ist, sich auf *rechenbare* Alphabete zu konzentrieren. Denn einerseits hilft dies bei der Konstruktion und Herleitung der Eigenschaften von Codes. Andererseits muss man ja auch bedenken, dass die Codierung und Decodierung in der Regel computergestützt erfolgt und vor allem für die Decodierung schnelle Verfahren erforderlich sind.

2.1.1 Der Körper der rationalen und reellen Zahlen

Mit den **rationalen Zahlen** $\mathbb{Q}$ und den **reellen Zahlen** $\mathbb{R}$ kann man z. B. gut rechnen. Man kann sie addieren und multiplizieren und es gelten dabei vernünftige Rechenregeln.

- $a + (b + c) = (a + b) + c$, $a(bc) = (ab)c$ **Assoziativgesetze**.
- $a + b = b + a, ab = ba$ **Kommutativgesetze**.
- $a(b + c) = ab + ac$ **Distributivgesetz**.
- Es gibt neutrale Elemente 0 und 1 bzgl. Addition und Multiplikation.
- Zu jedem Element a gibt es ein inverses Element bzgl. der Addition, nämlich $-a$, und
- zu jedem Element $b \neq 0$ gibt es ein inverses Element bzgl. der Multiplikation, nämlich b^{-1}.

Man nennt solche Mengen in der Mathematik einen **Körper** (engl. **Field**).

2.1.2 Endliche Körper

Sei dazu p eine Primzahl und $\mathbb{Z}_p$ die Reste modulo p. Wenn klar ist, dass wir uns in $\mathbb{Z}_p$ bewegen, lassen wir auch den Zusatz „mod p“ weg und schreiben $\mathbb{Z}_p = \{0, 1, \ldots, p-1\}$. Ist auch $\mathbb{Z}_p$ ein Körper? Alle Rechenregeln im Körper übertragen sich trivialerweise von $\mathbb{R}$ bzw. $\mathbb{Q}$ auf die Reste $\mathbb{Z}_p$, nur beim multiplikativen Inversen muss man etwas aufpassen. Sei dazu b eine natürliche Zahl mit $0 < b < p$. Da p eine Primzahl ist, sind b und p teilerfremd und es gibt ganze Zahlen r, s mit $rb+sp = 1$. Das folgt aus dem **euklidischen Algorithmus**, mithilfe dessen man den größten gemeinsamen Teiler zweier Zahlen als deren Vielfachsumme darstellen kann. Lesen wir diese Gleichung in $\mathbb{Z}_p$, so folgt, $rb = 1$ und $0 \neq b \in \mathbb{Z}_p$ hat ein multiplikatives Inverses, nämlich $b^{-1} = r$.

Also ist $\mathbb{Z}_p$ ein **Körper** mit p Elementen. Insbesondere ist $\mathbb{Z}_2 = \{0, 1\}$ der Körper der **binären Zahlen** (**Bits**) und $\mathbb{Z}_3 = \{0, 1, -1\}$ der Körper der **ternären Zahlen**.

Mit $\mathbb{Z}_p$ kann man sehr effektiv rechnen, da die Division durch p mit Rest und allgemeiner der euklidische Algorithmus effizient implementierbar sind.

Es gibt sogar zu jeder Primzahlpotenz p^m einen Körper K mit $q = p^m$ Elementen. Für $q = p$ haben wir den Körper eben explizit konstruiert, für echte p-Potenzen ist das nicht ganz so einfach. Wir werden erst wieder in Abschn. 5.1 darauf zurückkommen. In den folgenden Abschnitten werden wir häufig von Körpern K mit q Elementen sprechen. Zum Verständnis und auch für die Beispiele reicht es aber im Moment völlig aus, dass man sich $K = \mathbb{Z}_p$ vorstellt.

Es stellt sich die naheliegende Frage, ob auch $\mathbb{Z}_m$ ein Körper für zusammengesetztes m ist, z. B. für $m = 10$ oder $m = 9$. Das stimmt leider nicht. Nehmen wir speziell $m = 10$, dann gilt $2 \neq 0$ und $5 \neq 0$ in $\mathbb{Z}_{10}$ gelesen. Hätte also 2 ein multiplikatives Inverses r in $\mathbb{Z}_{10}$, so würde $2r = 1$ gelten. Wir multiplizieren die Gleichung mit $5 \neq 0$ und erhalten $5 = 5 \cdot 1 = 5(2r) = 10r = 0r = 0$, ein Widerspruch. Man nennt solche Mengen ohne multiplikatives Inverses einen kommutativen **Ring**. Wir werden Ringe jedoch nicht weiter betrachten.

Jetzt sind wir aber fast schon dort, wo wir hinwollten. Statt eines abstrakten endlichen Alphabets A für Blockcodes können endliche Körper K genommen werden. Da gibt es kleine, z. B. $\{0, 1\}$, beliebig große, aber eben immer nur solche von der Größe einer Primzahlpotenz. Aber Blockcodes sind ja n-Tupel über dem Alphabet $A = K$. Also sollten wir uns auch n-Tupel über K näher anschauen.

2.1.3 K^n als Vektorraum über dem endlichen Körper K

Über dem Körper $\mathbb{R}$ beispielsweise können wir Vektorräume bilden, den bekannten dreidimensionalen euklidischen Vektorraum $\mathbb{R}^3$, mit dem wir unsere Umgebung wahrnehmen und der eindimensionale Unterräume (Geraden) und zweidimensionale Unterräume (Ebenen) hat. Oder etwas abstrakter den Vektorraum der n-Tupel $\mathbb{R}^n$, mit dem manche Leser sicher auch schon etwas Erfahrung gesammelt haben. Warum also nicht einfach unsere Blockcodes als Teilmengen von K^n auffassen und für K^n alles nachmachen, was wir für $\mathbb{R}^n$ schon kennen?

Wir fassen also K^n als **Vektorraum** auf mit

- komponentenweiser Addition von Vektoren

$$(v_1, \ldots, v_n) + (w_1, \ldots, w_n) = (v_1 + w_1, \ldots, v_n + w_n)$$

und

- komponentenweiser Multiplikation mit Elementen von K, nämlich

$$c(v_1, \ldots, v_n) = (cv_1, \ldots, cv_n).$$

Ein **Unterraum** U von K^n ist eine Teilmenge, die unter den eben genannten Vektorraumoperationen abgeschlossen ist.

K^n hat viele Unterräume, z. B. eindimensionale Geraden und zweidimensionale Ebenen, wobei man bei der geometrischen Interpretation etwas vorsichtig sein muss. Jedes Objekt ist nämlich endlich, so hat eine Gerade genau $|K| = q$ Punkte, die Gerade mit Richtungsvektor $(1, 0, \ldots, 0)$ besteht z. B. genau aus der Punktmenge $\{(c, 0, \ldots, 0) | c \in K\}$.

Besonders wichtig ist aber die Tatsache, dass man für jeden Unterraum U von K^n Basisvektoren finden kann.

Basisvektoren sind eine Menge von Vektoren $\{u_1, \ldots, u_k\}$ im Unterraum U, die jeden Vektor im Unterraum U als K-**Linearkombination erzeugen**, selbst aber **linear unabhängig** sind. Man bezeichnet die Menge $\{u_1, \ldots, u_k\}$ auch kurz als **Basis**. Es gilt also:

- Zu jedem u aus dem Unterraum U gibt es Elemente $c_1, \ldots, c_k \in K$ mit $u = c_1u_1 + \ldots + c_ku_k$.
- Ist $d_1u_1 + \ldots + d_ku_k = 0$ für $d_i \in K$, so folgt, $d_1 = \ldots = d_k = 0$.

Eine Basis ist jedoch durch U *nicht* eindeutig bestimmt. Geometrisch kennt man das ja etwa von Ebenen im $\mathbb{R}^3$, die man auf viele Möglichkeiten mit zwei Vektoren aufspannen kann.

Die Anzahl der Basisvektoren k ist durch U eindeutig bestimmt. Man bezeichnet sie als **Dimension** $k = \dim(U)$ des Unterraums.

Diese Eigenschaften von Vektorräumen über einem Körper K, die ja für $\mathbb{R}^3$ bzw. $\mathbb{R}^n$ weitgehend bekannt sein sollten, erfordern bei ihrer Herleitung die Existenz von multiplikativen inversen Elementen in K. Weil im Folgenden besonders wichtig, wollen wir uns hier kurz überlegen, warum es bei endlichen Körpern K in Unterräumen $U \subseteq K^n$ stets Basisvektoren gibt und deren Anzahl eindeutig bestimmt ist.

Seien also $\{v_1, \ldots, v_m\}$ erzeugende Vektoren von U mit kleinstmöglicher Anzahl m. Solche gibt es jedenfalls, da $U \subseteq K^n$ endlich ist. Wären diese v_i nicht linear unabhängig, so könnte man $d_1 v_1 + \ldots + d_m v_m = 0$ schreiben mit – sagen wir – $d_1 \neq 0$. Damit ist aber $v_1 = -d_1^{-1}(d_2 v_2 + \ldots + d_m v_m)$ und folglich kann jedes Element aus U bereits als K-Linearkombination der Vektoren $\{v_2, \ldots, v_m\}$ geschrieben werden, entgegen der minimalen Wahl von m. Also ist $\{v_1, \ldots, v_m\}$ eine Basis von U.

Seien nun $\{u_1, \ldots, u_k\}$ und $\{v_1, \ldots, v_m\}$ zwei Basen von U. Dann lässt sich $v_1 \neq 0$ schreiben als $v_1 = c_1 u_1 + \ldots + c_k u_k$ mit – sagen wir wieder – $c_1 \neq 0$. Hieraus folgt $u_1 = c_1^{-1}(v_1 - c_2 u_2 - \ldots - c_k u_k)$ und somit erzeugen auch die Vektoren $\{v_1, u_2, \ldots, u_k\}$ den Unterraum U. Sei nun $d_1 v_1 + d_2 u_2 + \ldots + d_k u_k = 0$. Setzen wir hier v_1 ein, so ergibt sich $d_1 c_1 u_1 + (d_1 c_2 + d_2) u_2 + \ldots + (d_1 c_k + d_k) u_k = 0$. Wegen der linearen Unabhängigkeit der u_i und $c_1 \neq 0$ folgt zunächst $d_1 = 0$ und damit auch für die restlichen $d_i = 0$. Dies zeigt, dass die $\{v_1, u_2, \ldots, u_k\}$ auch linear unabhängig sind und daher eine Basis von U bilden. Man hat also u_1 durch v_1 ausgetauscht (sog. **Steinitz'scher Austauschsatz**). Dieses Argument wiederholt angewandt – also formal ein Induktionsbeweis – zeigt, dass m der u_i durch die v_i ausgetauscht werden können, also insbesondere $m \leq k$. Dreht man diese Argumentation um, so kann man genauso gut k der v_j durch die u_j austauschen, also gilt auch $k \leq m$. Insgesamt ist damit $k = m$ gezeigt und beide Basen haben gleich viele Elemente, d. h. der Begriff der Dimension macht Sinn.

Für n-Tupel über Ringen (statt Körpern) gelten die Aussagen über Basis und Dimension nicht. Das Rechnen mit Basen und Dimensionen ist jedoch wichtig im Zusammenhang mit der Konstruktion von guten Codes. Das ist der Grund, warum man sich auf Vektorräume über Körpern konzentriert.

2.1.4 Das Skalarprodukt auf K^n für endliche Körper K

Jetzt gehen wir noch einen letzten Schritt weiter und machen auch das bekannte **Skalarprodukt** im $\mathbb{R}^3$ bzw. $\mathbb{R}^n$ nach.

Wir definieren für $v = (v_1, \ldots, v_n)$ und $w = (w_1, \ldots, w_n) \in K^n$ das **Skalarprodukt** als $\langle v, w \rangle = v_1 w_1 + \ldots + v_n w_n$.

Wie man vom Skalarprodukt im $\mathbb{R}^n$ weiß oder aber sofort nachrechnet, gelten die folgenden Rechenregeln für $u, v, w \in K^n$ und $c \in K$.

- $\langle u + v, w \rangle = \langle u, w \rangle + \langle v, w \rangle$
- $\langle cu, v \rangle = c \cdot \langle u, v \rangle$
- $\langle u, v \rangle = \langle v, u \rangle$

- $\langle 0, v\rangle = 0$
- Ist $\langle u, v\rangle = 0$ für alle v, so ist $u = 0$, sog. **Regularität**.

Im euklidischen Raum ist man gewöhnt, dass man mit dem Skalarprodukt überprüfen kann, ob zwei Vektoren senkrecht aufeinander stehen, nämlich genau dann, wenn ihr Skalarprodukt gleich 0 ist. Die geometrische Anschauung des *Senkrechtstehens* muss man aber leider bei endlichen Körpern über Bord werfen. Viel schlimmer noch: Vektoren können *auf sich selbst senkrecht stehen*. Nimmt man beispielsweise $(1, 1) \in (\mathbb{Z}_2)^2$, so gilt, $\langle (1, 1), (1, 1)\rangle = 1 + 1 = 0$.

Wir verzichten hier auf konkrete Beispiele von Unterräumen über endlichen Körpern und deren Skalarprodukt, da uns diese ohnehin in den folgenden Abschnitten als Codes dauernd begegnen werden.

2.2 Grundlagen linearer Codes

2.2.1 Was versteht man unter linearen Codes?

Sei K ein endlicher Körper und $|K| = q$. Ein Blockcode C mit Alphabet K und Länge n heißt **linearer Code**, wenn C ein Unterraum des Vektorraums K^n ist. Hat C die Dimension $\dim(C) = k$ und Minimalabstand $d(C) = d$, so heißt C ein **$[n, k, d]_q$-Code**.

Ist q aus dem Zusammenhang heraus klar oder spielt d bei der aktuellen Überlegung keine Rolle, so spricht man auch kurz von $[n, k, d]$- oder $[n, k]$-Codes. Der Code C hat damit $|C| = q^k$ Elemente, sog. **Codewörter**. Das heißt nicht unbedingt, dass jedes Codewort mit einer Informationseinheit aus B belegt sein muss. Zur Erinnerung, die **Codierfunktion** ist eine eineindeutige Zuordnung „Element von B“ $\rightarrow$ „Codewort in C“, die jedoch nicht unbedingt jedes Codewort erreichen muss. Man sagt dazu **injektiv**, aber nicht notwendig **surjektiv**. Zum Beispiel muss im ISBN-Code nicht jede mögliche Zahl im Code durch eine real existierende Buchnummer belegt sein, denn es müssen ja auch noch Nummern für Neuerscheinungen frei sein. In jedem Fall muss aber allgemein $|B| \leq |C| = q^k$ gelten. Wie man am besten eine solche Codierfunktion wählt, stellen wir vorläufig bis zu Abschn. 2.5 zurück.

2.2.2 Die Rate linearer Codes

Wir hatten den Begriff der Rate für Blockcodes im Zusammenhang mit dem Satz von Shannon bislang nur intuitiv definiert. Die Rate ist ein Maß für die Redundanz, je höher

nämlich die Rate ist, desto geringer ist die Redundanz. Für lineare Codes vereinfacht sich das.

Die **Rate** linearer Codes ist gegeben als Verhältnis von Dimension k und Länge n, also $\textbf{Rate} = \boldsymbol{k/n}$.

Rate k/n und Minimalabstand d sind gegenläufig. Größerer Minimalabstand (d. h. bessere Fehlerkorrektureigenschaft) bedeutet gleichzeitig geringere Rate (d. h. größere Redundanz).

Das **Ziel der Codierungstheorie** kann man also wie folgt präzisieren. Man finde möglichst lineare Codes von

- für die jeweilige Anwendung praktikabler Länge n mit
- möglichst großem Minimalabstand d und
- möglichst großer Rate k/n, für die es aber auch
- schnelle und effiziente Codier- und Decodierverfahren gibt.

Die Konstruktionsmethode für *gute* Codes im Zusammenhang mit der **Gilbert-Schranke** liefert zwar Codes mit großer Rate, aber eben leider wenig handhabbare nichtlineare Codes. Wir werden uns im Folgenden stets auf lineare Codes beschränken. Für diese lassen sich z. B. der Hamming-Abstand und damit der Minimalabstand einfacher berechnen.

2.2.3 Gewicht und Minimalgewicht

Sei K ein endlicher Körper, $d(\ ,\ )$ der Hamming-Abstand auf K^n und C ein linearer Code in K^n.

- Man nennt $wt(v) = d(v, 0) = |\{i \,|v_i \neq 0\}|$ das **Gewicht** von $v = (v_1, \ldots, v_n) \in K^n$.
- Ist $C \neq \{0\}$, so heißt $wt(C) = \min\{wt(c)|0 \neq c \in C\}$ das **Minimalgewicht** von C. Für $C = \{0\}$ setzt man sinngemäß $wt(C) = 0$.

2.2.4 Minimalabstand bei linearen Codes

Mit den obigen Bezeichnungen gilt:

- $d(v, v') = d(v + w, v' + w)$ für alle $v, v', w \in K^n$, sog. **Translationsinvarianz**,
- $d(C) = wt(C)$, d. h. das Minimalgewicht ist gleich dem Minimalabstand.

Die erste Aussage ist sehr einfach einzusehen, da sich v und v' an genau denselben Stellen unterscheiden wie $v + w$ und $v' + w$. Die zweite Aussage hingegen ergibt sich aus

$$\begin{aligned} d(C) &= \min\{d(v, w) | v, w \in C,\ v \neq w\} = \min\{d(v - w, 0) | v, w \in C,\ v \neq w\} \\ &= \min\{d(u, 0) | u \in C,\ u \neq 0\} = wt(C). \end{aligned}$$

Für lineare Codes reduziert sich daher die Bestimmung des Minimalabstands auf die des Minimalgewichts, also von $|C|(|C|-1)/2$ Abstandsbestimmungen auf $|C|-1$ Gewichtsbestimmungen.

2.2.5 Der Träger

Im Zusammenhang mit dem Gewicht eines Vektors ist manchmal auch der Begriff des Trägers hilfreich. Sei also K ein endlicher Körper und $v = (v_1, \ldots, v_n) \in K^n$. Unter dem **Träger** von v versteht man die Menge $Tr(v) = \{i | v_i \neq 0\}$, also genau die Menge der Indizes von v mit Koordinate $v_i \neq 0$. Es gilt dabei $wt(v) = |Tr(v)|$. Den Begriff des Trägers benutzt man häufig dann, wenn man zwei Vektoren v und w addieren will. Es gilt nämlich:

- $wt(v + w) \geq wt(v) + wt(w) - 2|Tr(v) \cap Tr(w)|$,
- $wt(v + w) = wt(v) + wt(w) - 2|Tr(v) \cap Tr(w)|$ für binäre Vektoren über $K = \mathbb{Z}_2$.

Am besten man macht sich das an der Visualisierung in Abb. 2.1 klar. Hier sind symbolisch die zwei Vektoren v und w aufgemalt und die Träger mit Sternen markiert.

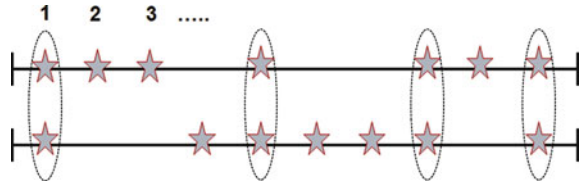


Abb. 2.1 Die Träger zweier Vektoren

2.2.6 Generatormatrix

Sei C ein $[n, k, d]$-Code über dem endlichen Körper K. Als Unterraum des K^n hat C eine Basis, sagen wir, $g_1 = (g_{11}, \ldots, g_{1n}), \ldots, g_k = (g_{k1}, \ldots, g_{kn})$. Dann heißt die Matrix

$$G = \begin{pmatrix} g_1 \\ \vdots \\ g_k \end{pmatrix} = \begin{pmatrix} g_{11} & \cdots & g_{1n} \\ \vdots & & \vdots \\ g_{k1} & \cdots & g_{kn} \end{pmatrix}$$

Generatormatrix (oder Erzeugermatrix) von C.

Die Generatormatrix hängt also von der Wahl der Basis ab. Wir werden im Folgenden keine Matrizenrechnung verwenden. Es hat sich aber nun mal eingebürgert, die Basis eines Codes in dieser kompakten Schreibweise als Generatormatrix aufzuschreiben.

2.2.7 Beispiel: Lineare Codes

Jetzt wird es sicher Zeit für ein paar Beispiele. Zunächst jedoch eine allgemeine Bemerkung: Wenn wir nun lineare Codes konstruieren, so müssen wir uns also Unterräume des K^n verschaffen. Dabei ist man häufig versucht, der Einfachheit halber einige der sog. **kanonischen Basisvektoren** $(1, 0, \ldots, 0)$, $(0, 1, 0, \ldots, 0), \ldots$ mit unterzubringen. Wenn man das macht, hat man schon verloren, denn diese drücken das Minimalgewicht des Codes ja runter auf 1. Dies ist der Grund, warum man hier und auch später recht knifflige Konstruktionen wählt.

Paritätsprüfungs-, Wiederholungs- und ISBN-Code
Wir kommen zunächst einmal auf einige unserer Eingangsbeispiele zurück. Der Paritätsprüfungscode, der Wiederholungscode und der ISBN-Code sind allesamt lineare Codes über $\mathbb{Z}_2$ bzw. $\mathbb{Z}_{11}$, da die Erweiterungsstellen jeweils durch lineare Gleichungen der übrigen Stellen ohne additive Konstanten beschrieben werden und $\mathbb{Z}_2$ bzw. $\mathbb{Z}_{11}$ jeweils Körper sind. Die Codes, bei denen modulo 10 bzw. 9 gerechnet wird, sehen zwar zunächst auch linear aus, sind es aber nicht, da sie über dem Ring $\mathbb{Z}_{10}$ bzw. $\mathbb{Z}_9$ gebildet werden.

Der Paritätsprüfungscode hat Parameter $[3, 2, 2]_2$ und Generatormatrix

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

Natürlich kann man auch allgemeiner ein binäres n-Tupel um ein Paritätsbit erweitern und erhält so einen $[n + 1, n, 2]_2$-Code.

Der Wiederholungscode hat Parameter $[6, 2, 3]_2$ und Generatormatrix

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

Der ISBN-Code hat Parameter $[10, 9, 2]_{11}$ und folgende Generatormatrix mit 10 Spalten und 9 Zeilen:

$$G = \begin{pmatrix} 1 & 0 & \dots & & 0 & 1 \\ 0 & 1 & 0 & \dots & 0 & 2 \\ \vdots & & & & & \vdots \\ 0 & 0 & \dots & 0 & 1 & 9 \end{pmatrix}.$$

IBAN-Code

Der IBAN-Code ist etwas diffiziler. Man berechnet dort die Prüfziffer bezogen auf „98“, sodass bei der Validierung 1 heraus kommt. Hätte man sie auf „97“ bezogen, so wäre dieser Code C linear gewesen und bei der Validierung wäre 0 herausgekommen.

Nehmen wir hierfür konkret das Beispiel DE für Deutschland. Hier ist die Kontoidentifikation 18-stellig, zusätzliche 4 Ziffern für „DE“ ergeben eine 22-stellige Zahl. Wie wir wissen, ist $\mathbb{Z}_{97}$ ein Körper, da 97 eine Primzahl ist. Wir betrachten $(\mathbb{Z}_{97})^{23}$ und lesen die einzelnen Stellen modulo 97, also in $\mathbb{Z}_{97}$. Der Code lässt sich dann beschreiben als

$$C = \{(c_1, \dots, c_{23}) | c_i \in \mathbb{Z}_{97},\ 10^{23}c_1 + 10^{22}c_2 + \dots + 10^3 c_{21} + 10^2 c_{22} + c_{23} = 0\}.$$

Bis auf die letzte Stelle c_{23} kommen bei allen anderen nur Werte zwischen 0 und 9 vor. Der Code hat Länge 23, Dimension 22 und als Generatormatrix

$$G = \begin{pmatrix} 1 & 0 & \dots & & 0 & r_{23} \\ 0 & 1 & 0 & \dots & 0 & r_{22} \\ \vdots & & & & & \vdots \\ 0 & 0 & \dots & 0 & 1 & r_2 \end{pmatrix}$$

mit 23 Spalten und 22 Zeilen, wobei sich r_i aus $10^i + r_i = 0$ berechnet.

Der kleine binäre Hamming-Code $Ham_2(3)$ mit Parameter $[7, 4, 3]_2$

Sei $C = \{(c_1, \dots, c_7) | c_i \in \mathbb{Z}_2,\ c_1 + c_2 + c_5 + c_7 = 0,\ c_3 + c_5 + c_6 + c_7 = 0,\ c_1 + c_4 + c_6 + c_7 = 0\}$. Dann hat C als Generatormatrix

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

Man sieht dies daran, dass die Zeilenvektoren von G einerseits die drei linearen Gleichungen in der Definition von C erfüllen und andererseits offenbar linear unabhängig sind. Wegen $\dim(C) = 4$ hat $|C|$ genau $2^4 - 1 = 15$ Codewörter $\neq 0$. Diese kann man alle noch im Kopf auf ihr Gewicht hin überprüfen und findet $wt(C) = 3$. Also ist C ein $[7, 4, 3]_2$-Code und heißt **kleiner binärer Hamming-Code** $Ham_2(3)$. Er gehört zu einer ganzen Familie von Codes, die wir in Abschn. 3.1 kennenlernen werden. Dort sehen wir auch einen Weg, wie man das Minimalgewicht des Codes bestimmt, ohne alle Codewörter einzeln zu überprüfen. Übrigens: $Ham_2(3)$ ist äquivalent zu dem Code, den wir in Abschn. 1.3 bereits als kleinen kombinatorischen Code explizit mit seinen 16 Elementen angegeben haben.

Der ternäre Golay-Code G_{11} mit Parameter $[11, 6, 5]_3$

Der Schweizer Elektroingenieur **Marcel Golay** (1902–1989) hat diesen und die in den nächsten Abschnitten beschriebenen, nach ihm benannten berühmten Codes 1949 publiziert. Er hat dafür je eine Generatormatrix angegeben. Wir haben eben nämlich gesehen, dass die Beschreibung eines Codes mittels Generatormatrix meist überschaubarer ist als die Definition in Mengenschreibweise. Hier ist eine Generatormatrix für den ternären Golay-Code G_{11} über $K = \mathbb{Z}_3$.

$$G = \begin{pmatrix} 1 & & & & & & 1 & 1 & 1 & 1 & 1 \\ & 1 & & & & & 0 & 1 & -1 & -1 & 1 \\ & & 1 & & & & 1 & 0 & 1 & -1 & -1 \\ & & & 1 & & & -1 & 1 & 0 & 1 & -1 \\ & & & & 1 & & -1 & -1 & 1 & 0 & 1 \\ & & & & & 1 & 1 & -1 & -1 & 1 & 0 \end{pmatrix}$$

Klar ist, dass der Code Länge 11, Dimension 6 und damit $3^6 - 1 = 728$ Codewörter $\neq 0$ hat. Das Minimalgewicht zu bestimmen, wird hier im Kopf schon schier unmöglich und auf dem Papier sehr mühsam. Ein kleines Rechenprogramm hilft oder ein Trick, den wir im nächsten Abschnitt sehen werden.

Ein kleiner Reed-Solomon-Code mit Parameter $[4, 3, 2]_5$

Als Beispiel für $K = \mathbb{Z}_5$ betrachten wir den linearen Code C über K mit Generatormatrix

$$G = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 \\ 1 & 4 & 4 & 1 \end{pmatrix}.$$

Da die Zeilen z_1, z_2 und z_3 von G offenbar linear unabhängig sind, hat C Dimension 3.

Wäre der Minimalabstand $d(C) \geq 3$, so wären mit zwei verschiedenen Codewörtern (c_1, c_2, c_3, c_4) und (c'_1, c'_2, c'_3, c'_4) auch die 4-Tupel $(c_1, c_2, 0, 0)$ und $(c'_1, c'_2, 0, 0,)$ verschieden. Das kann natürlich nicht sein wegen $|C| = 5^3$. Etwas Rechnung zeigt außerdem, dass

keiner der kanonischen Basisvektoren $(1,0,0,0),\ldots,(0,0,0,1)$ sich als Linearkombination von z_1 bis z_3 darstellen lässt. Damit sind diese nicht in C und folglich ist $d(C)=2$. Also ist C ein $[4,3,2]_5$-Code. Er gehört zur Familie der **Reed-Solomon-Codes**, die wir allerdings erst in Abschn. 5.2 genauer betrachten werden.

2.3 Erweiterte und duale Codes

Im Zusammenhang mit linearen Codes ist es häufig hilfreich, die folgenden beiden aus einem linearen Code abgeleiteten Codes zu betrachten.

2.3.1 Der erweiterte Code

Sei C ein linearer $[n,k,d]$-Code über dem endlichen Körper K. Man bezeichnet mit

$$C^{\wedge}=\{(c_1,\ldots,c_n,c_{n+1})|(c_1,\ldots,c_n)\in C,\ c_1+\ldots+c_n+c_{n+1}=0\}$$

den **erweiterten Code** zu C.

Der Code $C^{\wedge}$ ist offensichtlich ein linearer $[n+1,k,d^{\wedge}]$-Code über K, wobei $d\leq d^{\wedge}\leq d+1$ gilt. Man erhält eine Basis von $C^{\wedge}$, indem man eine Basis von C um die sog. **Paritätsziffer** an Position $n+1$ erweitert. Ist C binär, so ist die Anzahl der Einsen in jedem Codewort aus $C^{\wedge}$ gerade und $C^{\wedge}$ hat folglich gerades Minimalgewicht. Ist also für einen binären Code d gerade, so ist $d^{\wedge}=d$, ist d ungerade, so ist $d^{\wedge}=d+1$. Jedenfalls ist $C^{\wedge}$ eine Konstruktion, die unserem Paritätsprüfungscode in den Eingangsbeispielen entspricht, nur dass hier ein beliebiger linearer Ausgangscode mit einer Paritätsziffer erweitert wird.

2.3.2 Der duale Code

Sei wieder C ein linearer $[n,k,d]$-Code über dem endlichen Körper K und $\langle\ ,\ \rangle$ das Skalarprodukt auf K^n, also $\langle(x_1,\ldots,x_n),(y_1,\ldots,y_n)\rangle=x_1y_1+\ldots+x_ny_n$. Dann heißt $C^{\perp}=\{v\in K^n|\langle v,c\rangle=0 \text{ für alle } c\in C\}$ der **duale Code** zu C.

$C^\perp$ hat natürlich ebenfalls Länge n und ist ein linearer Code wegen der Linearität des Skalarprodukts. Aber Vorsicht: Die anschauliche Vorstellung, dass Vektoren aus $C^\perp$ *senkrecht* auf Vektoren aus C stehen, wie wir das in $\mathbb{R}^n$ gewohnt sind, gilt hier nicht. Es gibt sogar Codes, für die $C = C^\perp$ gilt; diese nennt man **selbstdual**. Wir kommen gleich zu einer wichtigen Eigenschaften von $C^\perp$.

Es gilt $\dim(C^\perp) = n - \dim(C) = n - k$.

Leider liegt diese Aussage nicht ganz auf der Hand. Wir wollen daher zunächst ein eher heuristisches Argument anführen. Sei dazu $v = (x_1, \ldots, x_n) \in C^\perp$ und $c_1 = (c_{11}, \ldots, c_{1n}), \ldots, c_k = (c_{k1}, \ldots, c_{kn})$ eine Basis von C. Nach Definition von $C^\perp$ gilt dann $0 = \langle v, c_i \rangle = x_1 c_{i1} + \ldots + x_n c_{in}$ für $i = 1, \ldots, k$. Wir haben es daher mit einem linearen Gleichungssystem über dem Körper K mit k Gleichungen, n Unbestimmten x_j und $k \leq n$ zu tun. Hier scheint ein kurzer Einschub – mit Analogie zu linearen Gleichungen über $\mathbb{R}$ – sinnvoll. Für $k < n$, also bei weniger Gleichungen als Unbestimmten, ergeben sich mehr als eine Lösung. Ein einfaches Beispiel hierzu ist *eine* Gleichung mit *zwei* Unbestimmten x, y über dem Körper $\mathbb{R}$, also $ax + by = 0$. Die Lösungsmenge ist dann eine Gerade in der (x, y)-Ebene, also ein eindimensionaler Unterraum des $\mathbb{R}^2$. Aber halt: Stimmt das wirklich? Voraussetzung dafür ist natürlich, dass nicht beide Koeffizienten a und b gleich 0 sind oder, etwas kunstvoller ausgedrückt, dass der eine Vektor $(a, b) \neq 0$ linear unabhängig ist. Hier ist die Verallgemeinerung dieser Tatsache: Wir betrachten nämlich die aus den Koeffizienten unseres Gleichungssystems zeilenweise gebildeten Vektoren $c_1 = (c_{11}, \ldots, c_{1n}), \ldots, c_k = (c_{k1}, \ldots, c_{kn})$. In unserem Fall wissen wir ja, dass die $c_1, \ldots, c_k$ als Basis von C linear unabhängig sind. Daher bildet die Lösungsmenge $(x_1, \ldots, x_n)$ des Gleichungssystems einen $(n - k)$-dimensionalen Unterraum des K^n. Anschaulich gesprochen ist das nicht verwunderlich: Bei der Berechnung der x_j hat man nämlich genau $n - k$ *Freiheitsgrade*. Also hat die Lösungsmenge $(x_1, \ldots, x_n)$ des Gleichungssystems, welche nach Konstruktion genau $C^\perp$ ist, die Dimension $n - k$.

Hier ist ein formaleres Argument, welches allerdings deutlich mehr lineare Algebra verwendet. Da nämlich die Basisvektoren $c_1, \ldots, c_k$ von C linear unabhängig sind, hat die zugehörige Generatormatrix maximalen Rang k. Daher ist die Abbildung $w \to (\langle w, c_1 \rangle, \ldots, \langle w, c_k \rangle)$ ein Epimorphismus von K^n auf K^k mit Kern $= C^\perp$. Dann folgt aus dem Homomorphiesatz $n - \dim(C^\perp) = \dim(K^n) - \dim(Kern) = \dim(K^k) = k$.

Als Folgerung erhalten wir $C = C^{\perp\perp}$.

Offensichtlich ist nämlich $C \subseteq C^{\perp\perp}$ und wegen $\dim(C^{\perp\perp}) = n - \dim(C^\perp) = n - (n - \dim(C)) = \dim(C)$ folgt $C = C^{\perp\perp}$.

2.3.3 Kontrollmatrix

Sei wieder C ein linearer $[n,k,d]$-Code über dem endlichen Körper K und $h_1 = (h_{11},\ldots,h_{1n}),\ldots,h_{n-k} = (h_{(n-k)1},\ldots,h_{(n-k)n})$ eine Basis von $C^\perp$. Dann ist die Matrix

$$H = \begin{pmatrix} h_1 \\ \vdots \\ h_{n-k} \end{pmatrix} = \begin{pmatrix} h_{11} & \ldots & h_{1n} \\ \vdots & & \vdots \\ h_{(n-k)1} & \ldots & h_{(n-k)n} \end{pmatrix}$$

eine Generatormatrix von $C^\perp$. Wegen $C = C^{\perp\perp}$ ist $v = (x_1,\ldots,x_n) \in K^n$ genau dann in C, wenn $\langle v, h_i\rangle = 0$ für $i = 1,\ldots,n-k$. Man nennt daher H auch **Kontrollmatrix** für C und die $\langle v, h_i\rangle = x_1h_{i1} + \ldots + x_nh_{in} = 0$ **Kontrollgleichungen** für v.

Die Kontrollgleichungen kann man auch in Vektorschreibweise für die Spaltenvektoren $s_1,\ldots,s_n$ der Kontrollmatrix H schreiben als $x_1s_1 + \ldots + x_ns_n = 0$. Es gelten mithin die beiden folgenden wichtigen Aussagen.

- Genau dann ist eine Matrix M Kontrollmatrix für C, wenn sie eine Generatormatrix von $C^\perp$ ist (nach Definition).
- Genau dann ist eine Matrix M Generatormatrix für C, wenn sie eine Kontrollmatrix von $C^\perp$ ist (wegen $C = C^{\perp\perp}$).

2.3.4 Zusammenhang zwischen erweitertem Code und dualem Code

Ist H eine Kontrollmatrix für den linearen $[n,k,d]_q$-Code C, d. h. eine Generatormatrix von $C^\perp$, so ist

$$H^\wedge = \begin{pmatrix} 1 & 1 & \ldots & 1 \\ & & & 0 \\ & H & & \vdots \\ & & & 0 \end{pmatrix}$$

eine Kontrollmatrix des erweiterten Codes $C^\wedge$, d. h. Generatormatrix von $(C^\wedge)^\perp$.

Man muss sich dazu überlegen, dass die Zeilenvektoren von $H^\wedge$ eine Basis von $(C^\wedge)^\perp$ bilden. Zunächst stimmt mal die Länge, denn $(C^\wedge)^\perp$ hat Länge $n+1$. Nach Voraussetzung sind die Zeilenvektoren von H linear unabhängig, da H Generatormatrix von $C^\perp$ ist. Wegen der einzigen 1 in der letzten Spalte von $H^\wedge$ ist aber auch die erste Zeile $(1,\ldots,1)$ von $H^\wedge$ linear unabhängig von den restlichen Zeilenvektoren. Also sind alle Zeilenvektoren linear unabhängig und $H^\wedge$ erzeugt folglich einen Code der Dimension $n-k+1$. Es gilt aber auch andererseits $\dim((C^\wedge)^\perp) = (n+1) - \dim(C^\wedge) = (n+1) - k = n-k+1$. Jetzt muss man sich nur noch überlegen, dass alle Zeilenvektoren von $H^\wedge$ Skalarprodukt 0 mit den Elementen des Codes $C^\wedge$ bilden. Dies ist aber klar für die erste Zeile $(1,\ldots,1)$ von $H^\wedge$, da dies ja genau die Paritätsbedingung in der Definition von $C^\wedge$ ist, und auch für die übrigen Zeilen nach Definition von H als Kontrollmatrix von C und wegen der 0 in der letzten Spalte.

2.3.5 Beispiel: Erweiterte und duale Codes

Kontrollmatrizen und -gleichungen spielen eine wichtige Rolle bei der Decodierung von linearen Codes, wie wir im nächsten Abschnitt sehen werden. Jetzt ist es aber sicherlich Zeit für einige ausführliche Beispiele.

Der duale ISBN-Code
Der duale Code des ISBN-Codes hat Länge 10, Dimension 1 und Generatormatrix

$$H = \begin{pmatrix} 10 & 9 & \ldots & 2 & 1 \end{pmatrix}.$$

Diese ist Kontrollmatrix des ISBN-Codes und liefert genau die Kontrollgleichung, über die der ISBN-Code definiert ist. Der duale Code hat zwar beeindruckendes Minimalgewicht $d = 10$, aber eine katastrophale Rate von $k/n = 1/10$.

Der kleine binäre Simplexcode $Sim_2(3)$ mit Parameter $[7,3,4]_2$
Wir haben im letzten Abschnitt eine Generatormatrix G_H für den kleinen binären Hamming-Code $Ham_2(3)$ erstellt, nämlich

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

Wir wollen nun den dualen Code zu $Ham_2(3)$ bestimmen. Das geht bei diesem kleinen Beispiel noch im Kopf. Wir suchen nämlich einen Code der Länge 7 und von Dimension 3, dessen Codewörter $(c_1,\ldots,c_7)$ alle mit den Zeilenvektoren von G_H Skalarprodukt

0 bilden. Wie man leicht nachrechnet, erfüllen die Zeilenvektoren der Matrix

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

diese Bedingung und sind linear unabhängig. Daher ist G_S eine Generatormatrix von $Ham_2(3)^{\perp}$. Auch hier ist die Anzahl $2^3 - 1 = 7$ der Codewörter $\neq 0$ noch übersichtlich genug, um zu sehen, dass der Code Minimalabstand $d = 4$ hat. Er heißt **kleiner binärer Simplexcode** $Sim_2(3)$ und gehört zu einer Serie von Codes, die wir ebenfalls in Abschn. 3.1 näher betrachten werden. Natürlich ist G_S wiederum Kontrollmatrix zu $Ham_2(3)$.

Der erweiterte ternäre Golay-Code G_{12} mit Parameter $[12, 6, 6]_3$

Wir betrachten den erweiterten Code G_{12} zum ternären Golay-Code G_{11}, den wir im letzten Abschnitt kurz angerissen haben. Dieser hat offenbar Generatormatrix

$$G^{\wedge} = \begin{pmatrix} 1 & & & & & & 1 & 1 & 1 & 1 & 1 & 0 \\ & 1 & & & & & 0 & 1 & -1 & -1 & 1 & -1 \\ & & 1 & & & & 1 & 0 & 1 & -1 & -1 & -1 \\ & & & 1 & & & -1 & 1 & 0 & 1 & -1 & -1 \\ & & & & 1 & & -1 & -1 & 1 & 0 & 1 & -1 \\ & & & & & 1 & 1 & -1 & -1 & 1 & 0 & -1 \end{pmatrix}$$

und ist daher ein sechsdimensionaler Code der Länge 12. Bilden wir nun die Skalarprodukte $\langle g_i, g_j \rangle$ der Zeilenvektoren $g_i (i = 1, \ldots, 6)$ von $G^{\wedge}$, d. h. einer Basis von G_{12}, so stellen wir fest, dass stets $\langle g_i, g_j \rangle = 0$ gilt; also ist G_{12} selbstdual. Wir wollen noch das Minimalgewicht von G_{12} bestimmen. Sei dazu $c = (c_1, \ldots, c_{12}) \in G_{12}$. Dann gilt einerseits $\langle c, c \rangle = 0$, da G_{12} selbstdual ist. Da andererseits die $c_i \neq 0$ alle entweder 1 oder -1 sind, gilt $0 = \langle c, c \rangle = c_1^2 + \ldots + c_{12}^2 = wt(c)1$ als Gleichung in $\mathbb{Z}_3$. Dies zeigt, dass $wt(c)$ ein Vielfaches von 3 sein muss. Man beachte nun, dass eine Linearkombination von zwei Zeilen von $G^{\wedge}$ ein Gewicht von genau 2 (aus den ersten sechs Positionen) plus mindestens 2 (aus den letzten sechs Positionen) hat, also insgesamt genau 6. Dies wiederum zeigt, dass eine Linearkombination von zwei Zeilen genau zweimal die 0 an den letzten sechs Positionen hat. Daraus folgt letztlich, dass eine Linearkombination von drei Zeilen von $G^{\wedge}$ ein Gewicht von genau 3 (aus den ersten sechs Positionen) plus mindestens 1 (aus den letzten sechs Positionen) hat, also insgesamt mindestens 6. Daher ist G_{12} ein selbstdualer $[12, 6, 6]_3$-Code.

Der ternäre Golay-Code G_{11} mit Parameter $[11, 6, 5]_3$

Es steht aus dem letzten Abschnitt noch die Bestimmung des Minimalgewichts von G_{11} aus. Da aber G_{11} aus G_{12} durch Weglassen der letzten Position und damit der Paritätsziffer entsteht, hat G_{11} ein Minimalgewicht von $d = 5$.

Der erweiterte kleine binäre Hamming-Code $Ham_2(3)^\wedge$ mit Parameter $[8, 4, 4]_2$
Hier ist zunächst wieder eine Generatormatrix des erweiterten Codes:

$$G_H^\wedge = \begin{pmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \end{pmatrix}.$$

Es handelt sich also um einen Code der Länge 8, von Dimension 4 und als binärer Code mit einem Minimalgewicht von $d^\wedge = 4$. Außerdem stellt man durch Skalarproduktbildung der Zeilen von $G_H^\wedge$ fest, dass der Code selbstdual ist. Insgesamt ist also $Ham_2(3)^\wedge$ ein selbstdualer $[8, 4, 4]_2$-Code.

Man sieht also, wie trickreich man manchmal vorgehen muss, um gute lineare Codes zu konstruieren. In den nächsten Abschnitten werden wir sehen, wie man ganze Serien solcher Codes definieren kann. Jetzt aber erst mal eine Pause beim Konstruieren. Wir wollen uns im nächsten Abschnitt den Verfahren und Hindernissen beim Decodieren widmen.

2.4 Maximum-Likelihood- und Syndromdecodierung

2.4.1 Hamming-Decodierung

Sei C vorübergehend nur ein Blockcode der Länge n und Minimalabstand d über dem Alphabet $A, |A| = q$. Wir haben uns bereits in Abschn. 1.3 mit den Kugeln vom Radius e um ein Codewort $c \in C$ beschäftigt, nämlich $B_e(c) = \{v \in A^n | d(v, c) \leq e\}$. Ist dabei $2e + 1 \leq d$, so sind jeweils zwei solche Kugeln disjunkt. Auf diese Weise haben wir den Begriff der e-fehlerkorrigierenden Codes herausgearbeitet. Treten nämlich bei jedem Codewort höchstens e Fehler auf, so kann man ein empfangenes Wort v korrekt zu dem einzigen Codewort im Abstand höchstens e korrigieren. Leider weiß man aber in der Praxis nicht, ob wirklich bei jedem ankommenden Wort v maximal e Fehler aufgetreten sind. Dennoch muss man – in der Regel ein Computer – automatisch reagieren.

> Man korrigiert v in *das* (oder wenn nicht eindeutig *ein*) Codewort, das an v am nächsten dran liegt, d. h. man sucht $c \in C$ mit $d(v, c) = \min\{(d(v, c') | c' \in C\}$. Man nennt dieses Verfahren auch **Hamming-Decodierung**.

Bei BD-Decodierung würde man zu Codewörtern im Abstand von mehr als e jedenfalls nicht korrigieren. Dabei wissen wir, dass das Verfahren das richtige Codewort liefert, falls höchstens e Fehler aufgetreten sind. Aber macht dieses Vorgehen auch wirklich Sinn? Lösen wir uns mal von unseren bisherigen Betrachtungen und fragen uns, auf welche Weise wir eigentlich gerne decodieren würden.

2.4.2 Maximum-Likelihood-Decodierung

Wir würden gerne ein empfangenes Wort v zu *dem* Codewort $c \in C$ korrigieren, für das die Wahrscheinlichkeit P am höchsten ist, dass v aus ihm stammt. Formaler heißt dies für die bedingte Wahrscheinlichkeit $P(\ |\ )$, dass wir $c \in C$ suchen mit $P(v|c) = \max\{P(v|c')|c' \in C\}$. Dieses Vorgehen wird als **Maximum-Likelihood-Decodierung** bezeichnet.

Obwohl intuitiv richtig, bringt uns das aber zunächst nicht viel weiter, da wir die bedingten Wahrscheinlichkeiten gar nicht kennen. Im Gegenteil, hier tritt noch eine andere Fragestellungen auf. Wenn z. B. ein deutscher Text gesendet wurde, so haben die Buchstaben im Text eine statistisch bekannte Häufigkeit. Wird also etwa ein V empfangen, das mit irgendeiner ausgefeilten Decodiermethode in den Buchstaben Q decodiert wird, ist denn dann die **Wahrscheinlichkeit**, dass V aus dem viel häufigeren E stammt, nicht **a priori** viel größer? Diese Frage stellt sich aber in der Praxis meist nicht. Denn der zu kanalcodierende String besteht aus einer langen Kette von – in der Regel binären – Ziffern, der die Quellencodierung nicht nur von Texten (d. h. Buchstaben), sondern auch von anderen, in Texten eingelagerten Objekten (Fotos, Grafiken etc.) darstellt. Dieser String wird bei der Kanalcodierung unabhängig von Sinninhalten jeweils in Blöcke zerschnitten und entsprechend dem verwendeten Code codiert.

Jetzt mag man einwenden, dass manchmal doch nur eine reine Textnachricht übertragen und jeder Buchstabe genau als einer dieser Blöcke quellencodiert wird. Hier noch ein weiteres Argument: Wie in Abschn. 1.1 beschrieben, werden Nachrichten häufig vorher mit Methoden der Kryptografie verschlüsselt. Ein wichtiges Merkmal dabei ist, dass statistische Häufigkeiten von Buchstaben *verschmiert* werden. Denn sonst wäre genau die Analyse der Häufigkeiten ein Angriffspunkt gegen die Chiffrierung. Zur Erinnerung: Die in der Einleitung erwähnte **Cäsar-Chiffre**, bei der jeder Buchstabe im Alphabet um drei Buchstaben nach hinten geschoben wird, kann man nämlich mit statistischen Auswertungen sofort knacken.

2.4.3 Hamming-Decodierung versus Maximum-Likelihood-Decodierung

Unsere Rettung aus dem Dilemma ist nun ein Resultat, das wiederum aus der **Informationstheorie** stammt und das im Wesentlichen besagt:

Für *in der Praxis gängige Kanäle* entspricht die **Hamming-Decodierung** dem eigentlich gewünschten **Maximum-Likelihood**-Verfahren.

Die Voraussetzung dieses Resultats und damit die Bedingung an den Kanal lauten konkret für ein Alphabet A mit $|A| = q$:

- Jedes $a \in A$ wird mit der gleichen Wahrscheinlichkeit $p < (q-1)/2$ verfälscht.
- Die $q-1$ Möglichkeiten, in die ein $a \in A$ verfälscht werden kann, sind alle gleich wahrscheinlich.

Für binäre Codes läuft dies also darauf hinaus, dass die Fehlerwahrscheinlichkeit pro Bit gleich $p < 1/2$ sein muss; man spricht dann von einem **binären symmetrischen Kanal**.

Wir wollen uns das Resultat für diesen Fall kurz überlegen. Dazu werde $c' = (c'_1, \ldots, c'_n) \in C$ gesendet und $v = (v_1, \ldots, v_n)$ empfangen mit Hamming-Abstand $d = d(v, c')$. Dann berechnet sich die bedingte Wahrscheinlichkeit $P(v|c')$ als

$$P(v|c') = \prod_i P(v_i|c'_i) = p^d(1-p)^{n-d} = (p/(1-p))^d(1-p)^n.$$

Wegen $p < \frac{1}{2}$ ist der Wert $a = p/(1-p)$ kleiner als 1 und somit die Funktion $f(x) = a^x$ monoton fallend, wie man aus der Analysis weiß. Daraus schließen wir, dass die Wahrscheinlichkeit $P(v|c) = \max\{P(v|c')|c' \in C\}$ genau für das (oder die) $c \in C$ ihr Maximum annimmt, für das (oder die) der Hamming-Abstand $d(v, c)$ minimal ist.

2.4.4 Syndromdecodierung

Wir sind damit zumindest so weit, dass unsere bisherigen Überlegungen sinnvoll waren und wir mit der Hamming-Decodierung weiterarbeiten können. Bei nichtlinearen Blockcodes bleibt beim Codieren daher nichts anderes übrig, als zu jedem empfangenen Wort v anhand der Liste aller Codewörter $c' \in C$ sämtliche Abstände $d(v, c')$ zu berechnen und ein c zu bestimmen, für das der Abstand minimal wird. Das ist aufwendig. Für lineare Codes geht das viel besser. Hierzu verwenden wir die Begriffe **Syndrom** und **Nebenklasse**, die wir jetzt erläutern wollen.

Sei wieder C ein $[n, k, d]$-Code über dem endlichen Körper K, $|K| = q$ mit der Kontrollmatrix

$$H = \begin{pmatrix} h_1 \\ \vdots \\ h_{n-k} \end{pmatrix} = \begin{pmatrix} h_{11} & \ldots & h_{1n} \\ \vdots & & \vdots \\ h_{(n-k)1} & \ldots & h_{(n-k)n} \end{pmatrix}.$$

Für einen Vektor $v \in K^n$ bezeichnet man den Vektor $s = (s_1, \ldots, s_{n-k})$ mit $s_i = \langle v, h_i \rangle$ als das **Syndrom** von v.

Wir erinnern uns, dass nach Definition der Kontrollmatrix v genau dann ein Codewort in C ist, wenn sein Syndrom gleich $(0, \ldots, 0)$ ist. Im Übrigen: Für ein beliebiges $h \in C^{\perp}$, also h eine Linearkombination der h_i, spricht man bei $\langle v, h \rangle$ von einem **Kontrollwert** für v.

Weiterhin nennt man $\{v + c | c \in C\}$ eine **Nebenklasse** von C in K^n.

Wir überlegen uns zunächst, dass K^n die disjunkte Vereinigung von q^{n-k} solcher Nebenklassen ist. Wählt man nämlich ein $u \in K^n$, welches nicht in der Nebenklasse zu v liegt, so ist die entsprechende Nebenklasse $\{u + c | c \in C\}$ disjunkt dazu. Sonst gäbe es nämlich $c_i \in C$ mit $v + c_1 = u + c_2$ und damit doch $u \in \{v + c | c \in C\}$. Auf diese Weise konstruieren wir uns wegen $|\{v + c | c \in C\}| = |C| = q^k$ sukzessive q^{n-k} solcher Nebenklassen.

Wegen $\langle v + c, h_i \rangle = \langle v, h_i \rangle$ für alle $c \in C$ ist das Syndrom für alle Elemente einer Nebenklasse gleich.

Man beachte aber, dass die Syndrome verschieden sind für verschiedene Nebenklassen. Sind nämlich für zwei Nebenklassen $\{v + c | c \in C\}$ und $\{u + c | c \in C\}|$ die Syndrome gleich, so folgt $\langle v - u, h_i \rangle = 0$ für alle i und die Kontrollgleichungen erzwingen $v - u \in C$, d. h. v und u liegen in derselben Nebenklasse.

Jetzt können wir die **Syndromdecodierung** beschreiben. In jeder der Nebenklassen suchen wir dazu einen Vektor f mit minimalem Gewicht $wt(f)$ innerhalb seiner Nebenklasse. Sofern f nicht eindeutig ist, wählen wir einen davon aus und nennen f einen **Nebenklassenführer**. Zu jedem Nebenklassenführer berechnen wir außerdem noch sein Syndrom $(\langle f, h_1 \rangle, \ldots, \langle f, h_{n-k} \rangle)$. Damit haben wir also eine Liste von insgesamt q^{n-k} Nebenklassenführern erstellt zusammen mit ihren jeweiligen Syndromen, die alle verschieden sind. Diese Rechnung ist zwar aufwendig, muss aber nur einmal für jeden Code gemacht werden und kann dann für jede weitere Decodierung genutzt werden.

Jetzt zur operativen Anwendung: Ein empfangenes Wort $v \in K^n$ muss ja in genau einer Nebenklasse liegen. Wir bilden daher sein Syndrom $(s_1, \ldots, s_{n-k}) = (\ldots, \langle v, h_i \rangle, \ldots)$ und suchen in unserer Liste den zugehörigen Nebenklassenführer f mit genau diesem Syndrom. Dann korrigieren wir v zu $c = v - f \in C$. Da f von minimalem Gewicht gewählt wurde, hat also das Codewort c kleinstmöglichen Abstand $d(v, c) = d(c + f, c) = d(f, 0) = wt(f)$ vom empfangenen Wort v.

Falls man mit der Syndrommethode BD-Decodierung (Bounded Distance) betreiben will, d. h. nur bis zur Fehlerkorrekturkapazität e eines Codes korrigieren möchte, so sepa-

riert man von vornherein alle Nebenklassenführer f, die ein Gewicht $wt(f) > e$ haben. Ermittelt man dann bei der Syndromdecodierung ein solches f, so korrigiert man nicht und markiert ggf. das empfangene Wort v entsprechend.

2.4.5 Beispiel: Syndromdecodierung

Ein kleiner binärer linearer Code mit Parameter [5,2,3]
Wir schauen uns nun ein konkretes Beispiel an, das übersichtlich genug ist, die einzelnen Schritte halbwegs im Kopf nachzuverfolgen. Sei dazu C ein binärer Code mit Kontrollmatrix

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

Wie man nachrechnet, ist $C = \{(0,0,0,0,0), (1,0,1,1,0,), (0,1,0,1,1), (1,1,1,0,1)\}$ ein $[5,2,3]_2$-Code. Er hat $2^{5-2} = 2^3 = 8$ Nebenklassen und ebenso viele Nebenklassenführer. Jede Nebenklasse hat $2^2 = 4$ Elemente, C selbst ist eine davon mit Nebenklassenführer $(0,0,0,0,0)$. Da $(1,0,0,0,0)$ nicht in C liegt, ist $\{(1,0,0,0,0), (0,0,1,1,0), (1,1,0,1,1), (0,1,1,0,1)\}$ eine weitere Nebenklasse mit Nebenklassenführer $(1,0,0,0,0)$ und Syndrom $(1,1,0)$. So bildet man sukzessive alle acht Nebenklassen. Ein bisschen Rechnung liefert letztlich folgende Liste von Nebenklassenführern mit zugehörigem Syndrom.

$$\begin{aligned}
f_1 &= (0,0,0,0,0) \quad \text{mit Syndrom} \quad (0,0,0).\\
f_2 &= (1,0,0,0,0) \quad \text{mit Syndrom} \quad (1,1,0).\\
f_3 &= (0,1,0,0,0) \quad \text{mit Syndrom} \quad (0,1,1).\\
f_4 &= (0,0,1,0,0) \quad \text{mit Syndrom} \quad (1,0,0).\\
f_5 &= (0,0,0,1,0) \quad \text{mit Syndrom} \quad (0,1,0).\\
f_6 &= (0,0,0,0,1) \quad \text{mit Syndrom} \quad (0,0,1).\\
f_7 &= (1,1,0,0,0) \quad \text{mit Syndrom} \quad (1,0,1).\\
f_8 &= (0,1,1,0,0) \quad \text{mit Syndrom} \quad (1,1,1).
\end{aligned}$$

Wird also etwa $v = (0,1,0,1,1)$ empfangen, so ist sein Syndrom $(0,0,0)$ und wir korrigieren v zu $v + f_1 = v$, d. h. v war bereits ein Codewort. Das Wort $v = (1,1,1,1,1)$ mit Syndrom $(0,1,0)$ korrigieren wir zum Codewort $v + f_5 = (1,1,1,0,1)$ und $v = (1,0,0,1,1)$ mit Syndrom $(1,0,1)$ zu $v + f_7 = (0,1,0,1,1)$.

Wegen Minimalabstand $d = 3$ ist C nur 1-fehlerkorrigierend. Bei BD-Decodierung würde man also im Falle der Nebenklassenführer f_7 und f_8 nicht korrigieren. Das empfangene Wort $v = (1,0,0,1,1)$ hat nämlich auch Abstand 2 zum Codewort $(1,0,1,1,0)$.

2.4.6 Komplexität: Hamming- und Syndromdecodierung

Ein wichtiges Kriterium für die Güte eines Decodierverfahrens ist seine **Komplexität**, d. h. die Größenordnung der benötigten elementaren arithmetischen Operationen und Vergleiche **asymptotisch** betrachtet für große Codelängen n. Komplexitäten benennen wir stets mit dem Symbol „$\sim$“. Man muss allerdings stets dabei beachten, dass die Komplexität nur einen Anhaltspunkt für die Schnelligkeit eines Verfahrens liefert. Mitentscheidend ist aber letztlich auch immer, wie die einzelnen Operationen in ihrem jeweiligen Kontext rechnerimplementierbar sind und wie viele Speicherzugriffe dafür benötigt werden.

Betrachten wir zunächst die Hamming-Decodierung eines $[n,k,d]_q$-Codes C. Für ein empfangenes Wort v müssen wir hierbei für jedes der q^k Codewörter $c \in C$ die Abstände $d(v,c)$ bestimmen; dies erfordert jeweils n Vergleiche bzw. Differenzbildungen mit Überprüfung auf $\neq 0$. Insgesamt benötigt die Hamming-Decodierung also nq^k elementare Operationen, d. h. die Komplexität ist $\sim nq^k$.

Bei der Syndromdecodierung muss man zunächst alle $\langle v, h_i \rangle$ berechnen, d. h. $2(n-k)n$ elementare Operationen ausgeführt. Das Syndrom muss man anschließend mit den Syndromen der q^{n-k} Nebenklassenführer f vergleichen, was insgesamt $(n-k)q^{n-k}$ elementare Operationen erfordert. Die Korrektur von v durch $v - f$ benötigt nur n elementare Operationen. Dies und den quadratischen Term $2(n-k)n$ kann man asymptotisch vernachlässigen. Daher ist bei Syndromdecodierung die Komplexität insgesamt $\sim (n-k)q^{n-k}$. Für Codes mit guter Rate k/n (d. h. k deutlich größer als $n/2$) ist somit die Syndromdecodierung wesentlich schneller.

Wir haben allerdings auch oben gesehen, dass die Syndromdecodierung selbst bei vergleichsweise kleinen Codes immer noch recht aufwendig ist. Um einen Eindruck vom benötigten Speicherplatzbedarf zu bekommen, nehmen wir mal einen binären Code C der Länge 70 und von Dimension 50, also mit Parameter $[70, 50]$. C hat dann 2^{20} Nebenklassenführer, jeder Nebenklassenführer mit seinem Syndrom benötigt 90 Bits. Daher ist der Speicherplatzbedarf $90 \cdot 2^{20}$ Bits $\sim$ 12 MBytes. Speichert man jedoch für die Hamming-Decodierung alle 2^{50} Codewörter, so hat man einen Speicherplatzbedarf von $70 \cdot 2^{50}$ Bits $\sim$ 9000 TBytes. Dies zeigt die Notwendigkeit, gezielt nach Codes zu suchen, die effiziente Decodieralgorithmen besitzen.

2.5 Codierung linearer Codes und Gesamtszenario

2.5.1 Generatormatrix in systematischer Form

Sei also wieder C ein linearer $[n,k,d]$-Code über dem endlichen Körper K mit Generatormatrix

$$G = \begin{pmatrix} g_1 \\ \vdots \\ g_k \end{pmatrix} = \begin{pmatrix} g_{11} & \cdots & g_{1n} \\ \vdots & & \vdots \\ g_{k1} & \cdots & g_{kn} \end{pmatrix}.$$

Wir nutzen nun folgende beiden Tatsachen.

- Mit $g_1, \ldots, g_k$ ist auch $x_1 g_1, g_2 + x_2 g_1, \ldots, g_k + x_k g_1$ (für $x_i \in K, x_1 \neq 0$) wieder eine Basis von C.
- Die Vertauschung der Stellen von C führt zu einem äquivalenten Code und ändert dessen Parameter nicht.

Da g_1 mindestens eine Komponente $\neq 0$ hat, können wir diese durch Stellenvertauschung und damit Übergang zu einem äquivalenten Code an die Position g_{11} bringen. Nun addieren wir Vielfache von g_1 zu den g_i derart, dass $g_{11} = 1$ und $g_{21} = \ldots = g_{k1} = 0$ gilt. Dieses Verfahren wenden wir nun auf die zweite Zeile der so entstandenen Matrix an und weiter sukzessive bis zur letzten Zeile. Auf diese Weise haben wir uns folgende Tatsache überlegt:

Jede Generatormatrix lässt sich in die sog. **systematische Form** (oder Standardform) $G_{SF} = (E_k|P)$ überführen. Hierbei bezeichnet E_k die Einheitsmatrix mit 1 auf der Diagonalen und 0 sonst und P ist eine Matrix mit k Zeilen und $n - k$ Spalten.

2.5.2 Systematische Codierung

Wie versprochen, kommen wir jetzt nochmals auf die Frage nach der Wahl einer bestmöglichen Codierfunktion zurück. Üblicherweise verwendet man nämlich für die zu codierenden Informationseinheiten B ebenfalls Tupel über dem gleichen Alphabet $A = K$, mit dem der lineare Code C gebildet wird. Hat C die Dimension k, so kann man damit offenbar Informationsumfänge $B \subseteq K^k$ codieren, wobei die Elemente aus B also k-Tupel über dem endlichen Körper K sind. Die Codierfunktion muss daher *einigen* der k-Tupel $(i_1, \ldots, i_k)$ über K jeweils ein Codewort zuordnen. Da man die Codierfunktion aber **algorithmisch** definieren möchte, ordnet man dabei in der Praxis üblicherweise *jedem* k-Tupel $(i_1, \ldots, i_k)$ ein Codewort $c \in C$ zu, unabhängig davon, ob dieses Tupel $(i_1, \ldots, i_k)$ wirklich mit einer Information aus B belegt ist oder nicht.

Zum Codieren benötigen wir eine eineindeutige Abbildungsvorschrift $K^k \to C \subseteq K^n$. Wir nutzen dazu unsere Generatormatrix G, bei der ja die Zeilenvektoren $g_1, \ldots, g_k$ eine Basis von C bilden. Als Codierfunktion wählen wir dann die Zuordnung $(i_1, \ldots, i_k) \to i_1 g_1 + \ldots + i_k g_k$.

Was ist aber nun der Vorteil der systematischen Form einer Generatormatrix?

Wenn die Generatormatrix von C in systematischer Form $G_{SF} = (E_k|P)$ vorliegt, so sieht die Codierfunktion konkreter so aus: $(i_1, \ldots, i_k) \rightarrow (i_1, \ldots, i_k, p_1, \ldots, p_{n-k})$. Dabei stehen also die **Informationsziffern** $i_1, \ldots, i_k$ an den ersten k Positionen und die **Paritätsziffern** $p_1, \ldots, p_{n-k}$ berechnen sich aus den Informationsziffern und den Einträgen der Matrix P. Eine solche Codierfunktion bezeichnet man als **systematisch**.

Eingangsbeispiele und ternärer Golay-Code aus Abschn. 2.2 sind systematisch.

2.5.3 Kanalcodierung und -decodierung – Gesamtszenario

Wir haben nun bereits einige Codes konkret konstruiert, haben uns überlegt, dass man mittels Generatormatrix in systematischer Form gut codieren kann, und haben auch schon Verfahren zur Decodierung gesehen. Deshalb wollen wir hier nochmals auf Abschn. 1.1 zurückkommen und das Gesamtszenario fehlerkorrigierender Codes besprechen. In unserem Modell besteht die Information aus k-Tupeln über einem Körper K (häufig binär), die dann in einen fehlerkorrigierenden Code C, nämlich einen Unterraum des Vektorraums der n-Tupel K^n $(n > k)$, kanalcodiert werden. Bevor aber wieder nach Empfang kanaldecodiert werden kann, d. h. Codewörter in $C \subseteq K^n$ wieder zurück nach K^k konvertiert werden können, müssen ja zunächst die Übertragungsfehler in einem empfangenen n-Tupel $v \in K^n$ korrigiert werden, d. h. v muss zu einem gültigen Codewort abgeändert werden. Auch diesen Schritt bezeichnet man lax als *Decodieren* und meint damit das Korrigieren von Fehlern mittels eines fehlerkorrigierenden Codes, also z. B. Hamming- und Syndromdecodierung. In Abb. 2.2 ist der Ablauf nochmals zusammengefasst.

Algorithmen zum Decodieren im Sinne der Fehlerkorrektur liegen nicht ganz so auf der Hand, wie wir gesehen haben und noch sehen werden. Das Kanalcodieren und auch -decodieren ist in der Regel durch die Definition des Codes, z. B. die Generatormatrix in systematischer Form, unmittelbar gegeben. Dennoch sucht man auch hier nach sehr effizienter Implementierung, um die Gesamtlaufzeit zu verkürzen. Optimal hierzu sind Schieberegister, auf die wir in Abschn. 6.2 näher eingehen werden.

Abb. 2.2 Gesamtszenario der Kanalcodierung und -decodierung

2.5.4 Beispiel: Gesamtszenario der Kanalcodierung und -decodierung

Der kleine binäre Hamming-Code $Ham_2(3)$
Zur Illustration des Ablaufs wieder ein Beispiel, nämlich der kleine binäre Hamming-Code $Ham_2(3)$, aber in systematischer Form. Hierzu betrachten wir zunächst einen zu unserem Simplexcode $Sim_2(3)$ äquivalenten Code mit folgender Generatormatrix:

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

Der duale Code $Ham_2(3)$ ist dann in systematischer Form mit Generatormatrix:

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

Damit wird die vierstellige Information (i_1, i_2, i_3, i_4) codiert in $(i_1, i_2, i_3, i_4, p_1 = i_1 + i_2 + i_4, p_2 = i_1 + i_3 + i_4, p_3 = i_2 + i_3 + i_4)$ mit den drei Paritätsbits p_1, p_2 und p_3. Abb. 2.3 zeigt die zugehörige digitale Schaltlogik.

Nach dem Senden werde das Wort $(i'_1, i'_2, i'_3, i'_4, p'_1, p'_2, p'_3)$ empfangen. Nun kommt unsere Fehlerkorrektur zum Tragen, z. B. das Syndromverfahren. Im Fall des kleinen binären Hamming-Codes berechnet sich das Syndrom (s_1, s_2, s_3) gemäß $s_1 = i'_1 + i'_2 + i'_4 + p'_1$, $s_2 = i'_1 + i'_3 + i'_4 + p'_2$ und $s_3 = i'_2 + i'_3 + i'_4 + p'_3$. Die Korrekturlogik des Syndromverfahrens entscheidet nun, ob bzw. welche der Informationsbits i'_1, i'_2, i'_3, i'_4 geändert werden müssen.

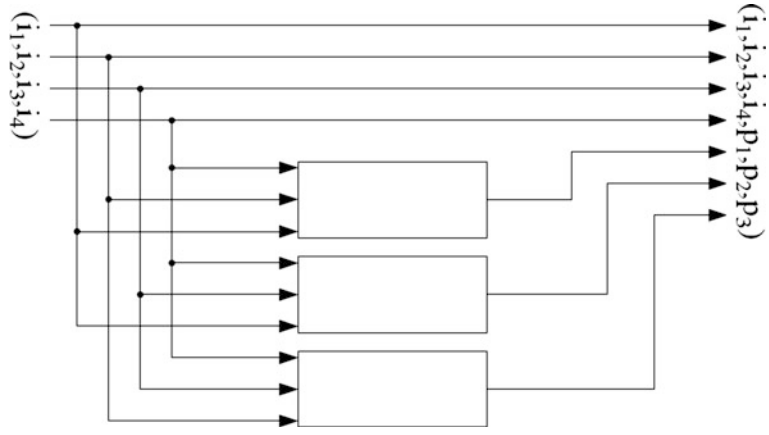


Abb. 2.3 Schaltlogik für die Codierung des kleinen binären Hamming-Codes

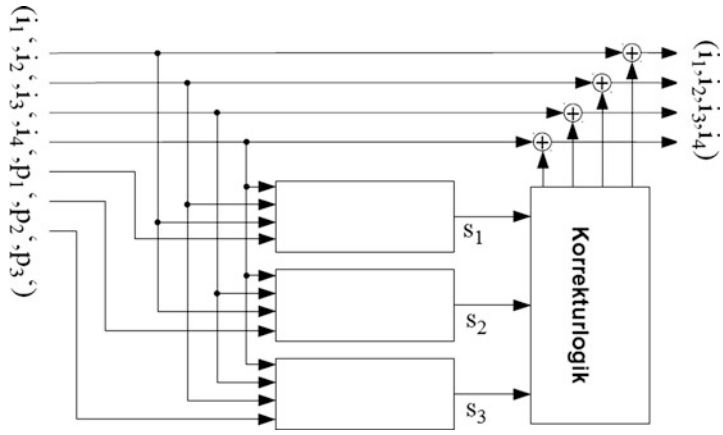


Abb. 2.4 Schaltlogik für die Decodierung des kleinen binären Hamming-Codes

Da der Hamming-Code Minimalabstand 3 hat, wird ja nur bei einem Fehler korrekt korrigiert. Das Kanaldecodieren ist dann wiederum sehr einfach, da durch die systematische Form der Generatormatrix die Informationsbits genau an den ersten vier Stellen ablesbar sind. Abb. 2.4 zeigt die zugehörige digitale Logik.

Hamming- und Golay-Codes

3

Hamming- und Simplexcodes Jetzt sind wir soweit, dass wir uns der ersten Familie von Codes zuwenden können: den von **Richard Hamming** 1950 publizierten berühmten **Hamming-Codes**, die er ursprünglich zum Erkennen von Fehlern bei Lochkarten erfand – eine Art *Durchbruch* in der Geschichte der Codierungstheorie. Den kleinsten Code dieser Serie kennen wir ja schon recht gut. Man konstruiert seine Codes, indem man eigentlich die dualen Codes und deren Generatormatrizen konstruiert, die **Simplexcodes**. Alle Hamming-Codes können einen Fehler sicher korrigieren. Wie üblich üben wir unser Konstruktionsverfahren an einfachen Beispielen.

Decodierung und Anwendungen der Hamming-Codes Zunächst wiederholen wir das wichtige Verfahren der Syndromdecodierung am Beispiel der Hamming-Codes. Danach lernen wir ein weiteres Decodierverfahren kennen, welches speziell für Hamming-Codes funktioniert, die sog. **Simplexdecodierung**. Am Ende stellen wir noch zwei Beispiele für Praxisanwendungen der Hamming-Codes vor – nämlich **ECC-Arbeitsspeicher** in Computern und das erste zivile Frequenzband L1 C/A der Satellitennavigation **GPS**.

Golay-Codes und perfekte Codes Den ternären Golay-Code kennen wir schon, jetzt konstruieren wir auch den nicht minder berühmten **binären Golay-Code**. Nun stellt sich natürlich die Frage nach dem Konstruktionsprinzip einer Familie von Golay-Codes – ähnlich dem der Hamming-Codes. Die Antwort darauf ist überraschend und erfordert einen kleinen Umweg über die sog. **Kugelpackungsschranke**, die wie alle Schranken wichtige Parameter eines Codes in Relation zueinander bringt. Codes, bei denen in der Kugelpackungsschranke die Gleichheit gilt, die also von Hamming-Kugeln eines festen Radius disjunkt überdeckt werden, nennt man **perfekt**. Der Klassifikationssatz aller perfekten Codes besagt, dass die Hamming-Codes sowie die beiden Golay-Codes die einzigen nichttrivialen perfekten Codes sind. Insbesondere kann man die Suche nach weiteren Golay-Codes getrost aufgeben.

O. Manz, *Fehlerkorrigierende Codes*, DOI 10.1007/978-3-658-14652-8_3

Wissenswertes und Kurioses rund um die Golay-Codes Golay-Codes haben eine erstaunliche Faszination ausgeübt. Schon die Originalarbeit von **Marcel Golay** aus dem Jahr 1949 von knapp einer Seite sagt im Wesentlichen nichts anderes aus als *heureka*. Es wurden aber in der Folge eine Unzahl von systematischeren Beschreibungen angegeben sowie Querverbindungen zu anderen mathematischen Gebieten hergestellt bis hin zum berühmten **Klassifikationssatz aller endlichen einfachen Gruppen**. Auf Basis des ternären Golay-Codes kann man sich auch als **Fußballwettkönig** versuchen. Fakt ist, dass die von den NASA-Sonden **Voyager 1 und 2** gesendeten Bilder von **Jupiter** und **Saturn** mittels binärem Golay-Code übertragen wurden. Auch in der klinischen Diagnostik gibt es Ansätze, die Golay-Codes in der **fotoakustischen Bildgebung** einzusetzen.

3.1 Hamming- und Simplexcodes

In den 1940er-Jahren war **Richard Hamming** Mitarbeiter am Forschungsinstitut AT&T Bell Labs. Er programmierte damals natürlich noch mit Lochkarten, die mittels elektromechanischen Relais in einen Großrechner eingelesen wurden. Die Lochkarten nutzten sich allerdings mit der Zeit ab, sodass beim Dateneinlesen hie und da Fehler auftraten. Zu normalen Büroarbeitszeiten war das kein großes Problem, da die Fehler von einem Operator der Bell Labs korrigiert werden konnten. Allerdings arbeitete Hamming oft außerhalb der Bürozeiten, sodass die sporadisch auftretenden Lesefehler nicht unmittelbar behoben werden konnten. Von dieser Situation und dem daraus resultierenden Mehraufwand nicht gerade begeistert, kam Hamming auf die Idee, einen Code zu entwickeln, mittels dessen der Rechner die Lesefehler von Lochkarten in bestimmtem Umfang selbstständig korrigieren konnte. Im Jahr 1950 publizierte er seine **Hamming-Codes**, u. a. den kleinen binären Hamming-Code der Länge 7, in seiner Arbeit mit dem Titel „Error Detection and Error Correction Codes“ [Ham]. Diese Codes finden noch bis heute die ein oder andere Anwendung. Wir wollen nun die Hamming-Codes beschreiben und beginnen mit der Konstruktion einer Matrix.

3.1.1 Die Matrix

Es sei k eine natürlich Zahl und K ein endlicher Körper mit $|K| = q$. Wir konstruieren nun eine Matrix M mit k Zeilen. Im Vektorraum K^k wählen wir dazu einen Vektor $v_1 \neq 0$ und schreiben diesen in die erste Spalte. Dann wählen wir einen weiteren Vektor $v_2 \neq 0$, der kein Vielfaches von v_1 ist, und schreiben diesen in die zweite Spalte. Dann wählen wir einen Vektor $v_3 \neq 0$, der nicht Vielfaches einer der vorhergehenden Spalten ist, und schreiben ihn in Spalte Nummer drei. So machen wir immer weiter. Da K^k nur endlich viele Vektoren enthält (nämlich q^k), finden wir irgendwann keinen solchen Vektor mehr. Sagen wir der letzte gefundene Vektor sei v_n. Wir haben somit eine Matrix $M = (v_1 v_2 \dots v_n)$ mit k Zeilen und n Spalten konstruiert. Wir stellen zunächst

fest, dass sich die Reihenfolge der Spalten von M rein zufällig ergeben hat. In jedem Schritt nämlich war die Wahl des nächsten v_i willkürlich, wir hätten z. B. auch genauso gut mit v_n anfangen können. Außerdem halten wir fest, dass bei jeder Wahl eines v_i wir auch ein von 0 verschiedenes Vielfaches av_i mit $0 \neq a \in K$ hätten wählen können. Das heißt, für jede Spalte hätte es genau $q-1$ solcher Kandidaten gegeben. Umgekehrt steht ein Vielfaches jedes Vektors $0 \neq v \in K^k$ in genau einer der Spalten von M, denn sonst hätte man ja M um diesen Vektor v erweitern müssen. Also ist $q^k - 1 = |K^k \setminus \{(0,\ldots,0)\}| = n/(q-1)$ und damit $n = (q^k-1)/(q-1)$. Noch etwas ist wichtig und leicht zu sehen. Natürlich sind Vielfache der kanonischen Basisvektoren unter den v_i. Wir können also ggf. nach Umsortierung der Spaltenvektoren von M davon ausgehen, dass $v_1 = (a_1, 0, \ldots, 0)$, $v_2 = (0, a_2, 0, 0, \ldots, 0), \ldots,$ $v_k = (0, \ldots, 0, a_k)$ mit $a_i \neq 0$ gilt, also

$$M = \begin{pmatrix} a_1 & 0 & \ldots & 0 & * & * \\ 0 & a_2 & & \vdots & & \\ \vdots & \ldots & & a_k & * & * \end{pmatrix}.$$

Betrachtet man jetzt mal zur Abwechslung die Zeilenvektoren $z_1, \ldots, z_k$ von M, so müssen diese folglich linear unabhängig sein. Nach dieser Vorbereitung kommen wir nun zu den angekündigten Codes.

3.1.2 Hamming- und Simplexcodes – die *Klassiker* unter den Codes

Es sei k eine natürliche Zahl und K ein endlicher Körper mit $|K| = q$.

Der bis auf Äquivalenz eindeutig bestimmte lineare Code mit Kontrollmatrix M heißt **Hamming-Code** $Ham_q(k)$. Seine Parameter sind $[n, n-k, 3]_q$ mit $n = (q^k-1)/(q-1)$.

Der bis auf Äquivalenz eindeutig bestimmte lineare Code mit Generatormatrix M heißt **Simplexcode** $Sim_q(k)$. Er ist der duale Code zum Hamming-Code $Ham_q(k)$ und hat die Parameter $[n, k, q^{k-1}]_q$.

Wir sammeln zuerst, was wir von den eben genannten Eigenschaften alles schon wissen und was wir noch zu überlegen haben.

Dass $n = (q^k-1)/(q-1)$ gilt, haben wir schon oben gesehen.

Die Eindeutigkeit bis auf Äquivalenz folgt aus der Willkür bei der Wahl der Reihenfolge und der Vielfachen der Spaltenvektoren v_i von M. Man muss hierbei beachten, dass die Multiplikation mit $0 \neq a \in K$ eine Permutation der Elemente von K bewirkt.

Die Zeilenvektoren einer Generatormatrix bilden ja stets eine Basis des zugehörigen Codes. Also bilden $z_1, \ldots, z_k$ eine Basis von $Sim_q(k)$. Folglich hat $Sim_q(k)$ Dimension k und $Ham_q(k)$ als dualer Code Dimension $n-k$.

Wir müssen uns also noch das Minimalgewicht beider Codes überlegen. Dabei ist es jeweils unerheblich, welche äquivalente Form der Codes wir betrachten.

Kommen wir zunächst zum Hamming-Code. Gäbe es also ein Codewort $c \neq 0$ in $Ham_q(k)$ vom Gewicht maximal 2. Dann können wir $c = (a, b, 0, \ldots, 0)$ mit $a \neq 0$ annehmen. Da M Kontrollmatrix für den Hamming-Code ist, folgt $\langle c, z_i \rangle = 0$ für die Zeilenvektoren z_i von M, konkret also $av_1 + bv_2 = 0$ und daher $v_1 = -a^{-1}bv_2$. Damit ist v_1 entweder 0 oder ein Vielfaches von v_2. Dieser Widerspruch zeigt, dass der Minimalabstand des Hamming-Codes mindestens 3 ist. Umgekehrt sind die Vektoren $(r, 0, \ldots, 0)$, $(0, s, 0, \ldots, 0)$ und $(t, t, 0, \ldots, 0)$ für $r = a_1, s = a_2$ und $t \neq 0$ unter den Spaltenvektoren v_i von M. Bei geeigneter Reihenfolge können wir annehmen, dass es sich dabei um v_1, v_2 und v_3 handelt. Damit ist $(r^{-1}, s^{-1}, -t^{-1}, 0, \ldots, 0)$ ein Codewort im Hamming-Code und der Minimalabstand ist genau 3.

Nun betrachten wir den Simplexcode. Sei dazu $c = (c_1, \ldots, c_n)$ ein Codewort $\neq 0$. Wir überlegen uns, dass dann immer $wt(c) = q^{k-1}$ gilt. Seien wieder $z_1, \ldots, z_k$ die Zeilenvektoren von M. Da M Generatormatrix des Simplexcodes ist, folgt $(c_1, \ldots, c_n) = c = b_1 z_1 + \ldots + b_k z_k = b_1(z_{11}, \ldots, z_{1n}) + \ldots + b_k(z_{k1}, \ldots, z_{kn})$ mit $b_i \in K$. Wir betrachten den Vektor $0 \neq b = (b_1, \ldots, b_k) \in K^k$ und den von b erzeugten eindimensionalen Unterraum U in K^k. Wir halten zunächst fest: Es ist $c_j = 0$ genau dann, wenn $b_1 z_{1j} + \ldots + b_k z_{kj} = 0$, und dies genau dann, wenn $v_j = (z_{1j}, \ldots, z_{kj}) \in U^\perp$. Wir wissen auch, dass $U^\perp$ Dimension $k-1$ hat. Damit enthält $U^\perp$ genau $(q^{k-1}-1)/(q-1)$ der Spaltenvektoren v_j von M. Somit hat $c = (c_1, \ldots, c_n)$ genau $(q^{k-1}-1)/(q-1)$ Koordinaten $\neq 0$ und das Gewicht $wt(c) = n-(q^{k-1}-1)/(q-1) = q^{k-1}(q-1)/(q-1) = q^{k-1}$ ist gleich für alle Codewörter ungleich 0.

3.1.3 Beispiel: Hamming- und Simplexcodes

Jetzt können wir natürlich eine ganze Reihe konkreter Beispiele konstruieren.

Binäre Hamming-Codes $Ham_2(k)$ und Simplexcodes $Sim_2(k)$
Für $k = 2$ ergibt sich ein $[3, 1, 3]$-Hamming-Code, nämlich ein trivialer Wiederholungscode der Form $\{(0, 0, 0), (1, 1, 1)\}$. Der entsprechende Simplexcode mit Parameter $[3, 2, 2]$ lautet $\{(0, 0, 0), (1, 1, 0), (1, 0, 1), (0, 1, 1)\}$ und ist daher genau der Paritätsprüfungscode, mit dem wir gestartet sind.

Für $k = 3$ erhält man die Parameter $[7, 4, 3]$ des kleinen binären Hamming-Codes, den wir bereits intensiv betrachtet haben. Den entsprechenden Simplexcode mit Parameter $[7, 3, 4]$ kennen wir ebenfalls schon. Zur Übung wollen wir trotzdem die Matrix M konstruieren. Wir müssen also eine Matrix mit 3 Zeilen, 7 Spalten und mit Einträgen aus $\mathbb{Z}_2$ bilden, bei der kein Spaltenvektor ein Vielfaches eines anderen Spaltenvektors ist. Da

die Reihenfolge unerheblich ist, nehmen wir als Spalten zunächst die Vektoren in $\mathbb{Z}_2^3$ mit genau einer Eins, dann die mit genau zwei Einsen und zuletzt $(1,1,1)$. Hier ist also die Matrix M – und damit der Simplexcode – in systematischer Form.

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

Wenn wir diese Matrix mit den Kontrollmatrizen des kleinen binären Hamming-Codes aus den Abschn. 2.3 und 2.5 vergleichen, so hatten die Spalten dort eine andere Reihenfolge, d. h. wir haben mit äquivalenten Codes gearbeitet. Die Gründe dafür sind, dass wir die Gestalt aus Abschn. 2.3 in Abschn. 3.3 für die Konstruktion des binären Golay-Codes G_{23} verwenden werden und dass die Gestalt aus Abschn. 2.5 den Hamming-Code in systematischer Form ergab.

Letztlich wollen wir zur Übung auch für den Fall $k = 4$ die Matrix M konstruieren. Jedenfalls wissen wir schon, dass der Hamming-Code Parameter $[15, 11, 3]$ und der entsprechende Simplexcode Parameter $[15, 4, 8]$ hat. Mittels des obigen Konstruktionsprinzips ergibt sich also M wieder in systematischer Form.

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

Im Übrigen gilt stets: Der erweiterte Code $Ham_2(k)^\wedge$ eines binären Hamming-Codes hat Minimalgewicht $d = 4$.

Ternäre Hamming-Codes $Ham_3(k)$ und Simplexcodes $Sim_3(k)$

Für $k = 2$ haben sowohl der ternäre Hamming-Code als auch der ternäre Simplexcode die Parameter $[4, 2, 3]_3$ und als Matrix M ergibt sich

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

Man stellt fest, dass M auch Generatormatrix von $Ham_3(2)$ ist und $Ham_3(2) = Sim_3(2)$ ist selbstdual.

Für $k = 3$ hat der ternäre Hamming-Code Parameter $[13, 10, 3]_3$ und der Simplexcode $[13, 3, 9]_3$. Hier ist wieder eine systematische Form für die Matrix M.

$$M = \begin{pmatrix} 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & -1 & 1 & 1 \\ 0 & 1 & 0 & 1 & -1 & 0 & 0 & 1 & 1 & 1 & 1 & -1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & -1 & 1 & -1 & 1 & 1 & 1 & -1 \end{pmatrix}$$

Hamming- und Simplexcodes $Ham_5(2)$ und $Sim_5(2)$ über $\mathbb{Z}_5$
Der Hamming-Code $Ham_5(2)$ hat Parameter $[6,4,3]_5$ und der Simplexcode $Sim_5(2)$ hat Parameter $[6,2,5]_5$. Für M ergibt sich die systematische Form

$$M = \begin{pmatrix} 1 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 2 & 3 & 4 \end{pmatrix}.$$

3.1.4 Güte der Hamming- und Simplexcodes

Simplexcodes besitzen zwar asymptotisch einen konstanten relativen Minimalabstand $d/n \sim q^{k-1}/q^k = 1/q$. Dagegen ist ihre Rate k/n asymptotisch schlecht, nämlich $\sim k/q^k$. Für binäre Codes und $k = 5$ ist die Rate schon ca. $1/6$, d. h. man muss außer der eigentlichen Information noch 5-mal den gleichen Datenumfang zusätzlich senden. Das ist für technische Anwendungen unbrauchbar, und Simplexcodes werden daher praktisch nicht angewandt. Bei Hamming-Codes liegt die Sache genau anders herum. Die Rate $(n-k)/n$ ist zwar sehr gut, für $k = 5$ z. B. schon über 0,8, und sie geht asymptotisch sogar gegen 1. Wegen $d = 3$ sind leider alle Hamming-Codes nur 1-fehlerkorrigierend. In der Praxis werden daher nur relativ kleine Hamming-Codes angewandt, wie wir im nächsten Abschnitt sehen werden.

3.2 Decodierung und Anwendungen der Hamming-Codes

3.2.1 Beispiel: Syndromdecodierung von Hamming-Codes

Hamming-Code $Ham_2(3)$
Natürlich kann man Hamming- und Simplexcodes auch wieder mit der Syndrommethode decodieren. Wir üben dies nun am Beispiel $Ham_2(3)$ in der äquivalenten Form des Abschn. 3.1. Hier ist die Liste der $2^3 = 8$ Nebenklassenführer der Länge 7 mit ihrem jeweiligen Syndrom.

Nebenklassenführer	Syndrom
$(0,0,0,0,0,0,0)$	$(0,0,0)$
$(1,0,0,0,0,0,0)$	$(1,0,0)$
$(0,1,0,0,0,0,0)$	$(0,1,0)$
$(0,0,1,0,0,0,0)$	$(0,0,1)$
$(0,0,0,1,0,0,0)$	$(1,1,0)$
$(0,0,0,0,1,0,0)$	$(1,0,1)$
$(0,0,0,0,0,1,0)$	$(0,1,1)$
$(0,0,0,0,0,0,1)$	$(1,1,1)$

Es werde das Wort $v = (1, 0, 1, 0, 1, 1, 1)$ empfangen. Das Syndrom von v ist $(1, 0, 0)$, also wird der Nebenklassenführer $f = (1, 0, 0, 0, 0, 0, 0)$ gewählt und v zu $c = v + f = (0, 0, 1, 0, 1, 1, 1)$ korrigiert.

Hamming-Code $Ham_3(2)$
Noch ein weiteres Beispiel, nämlich $Ham_3(2)$ in der äquivalenten Form des Abschn. 3.1. Hier sind wieder die $3^2 = 9$ Nebenklassenführer der Länge 4 mit ihrem jeweiligen Syndrom.

Nebenklassenführer	Syndrom
$(0, 0, 0, 0)$	$(0, 0)$
$(1, 0, 0, 0)$	$(1, 0)$
$(0, 1, 0, 0)$	$(0, 1)$
$(0, 0, 1, 0)$	$(1, 1)$
$(0, 0, 0, 1)$	$(1, -1)$
$(-1, 0, 0, 0)$	$(-1, 0)$
$(0, -1, 0, 0)$	$(0, -1)$
$(0, 0, -1, 0)$	$(-1, -1)$
$(0, 0, 0, -1)$	$(-1, 1)$

Es werde das Wort $v = (1, -1, 1, 1)$ empfangen. Das Syndrom von v ist $(0, -1)$, also wird der Nebenklassenführer $f = (0, -1, 0, 0)$ gewählt und v zu $c = v - f = (1, -1, 1, 1) - (0, -1, 0, 0) = (1, 0, 1, 1)$ korrigiert.

3.2.2 Simplexdecodierung von Hamming-Codes

Für Hamming-Codes gibt es aber ein anderes Decodierverfahren, die **Simplexdecodierung**. Hierzu nutzt man die spezielle Form des Simplexcodes. Sei also wieder M dessen Generatormatrix und damit die Kontrollmatrix eines Hamming-Codes $Ham_q(k)$ über dem Körper K mit Zeilenvektoren z_i für $i = 1, \ldots, k$. Es werde das Wort $v = (x_1, \ldots, x_n)$ empfangen. Wir betrachten wieder das Syndrom $s = (\langle v, z_1\rangle, \ldots, (\langle v, z_k\rangle)$ von v. Ist $s = 0$, so ist $v \in Ham_q(k)$ und v muss nicht korrigiert werden.

Ist $s \neq 0$, so muss nach Konstruktion von M der Vektor $s \in K^k$ ein von 0 verschiedenes Vielfaches genau einer der Spalten von M sein. Ist dies die j. Spalte v_j von M, so gilt also $s = av_j$ mit $0 \neq a \in K$. Wir korrigieren dann $v = (x_1, \ldots, x_n)$ zu $c = (x_1, \ldots, x_j - a, \ldots, x_n) \in Ham_q(k)$. Im binären Fall muss man also einfach das Bit an der Stelle j ändern.

3.2.3 Beispiel: Simplexdecodierung von Hamming-Codes

Wir nutzen wieder die äquivalente Form der Hamming-Codes aus Abschn. 3.1.

Hamming-Code $Ham_2(3)$
Es werde $v = (1,0,1,0,1,0,1)$ empfangen. Dann ist $s = (1,1,1)$ und dies ist die 7. Spalte von M. Wir korrigieren also v zu $c = (1,0,1,0,1,0,0) \in \mathrm{Ham}_2(3)$.

Hamming-Code $Ham_3(2)$
Es werde $v = (0,1,-1,0)$ empfangen. Dann ist $s = (-1,0)$ und damit ist s das (-1)-Fache der 1. Spalte von M. Wir korrigieren also v zu $c = (0-(-1),1,-1,0) = (1,1,-1,0) \in Ham_3(2)$.

Hamming-Code $Ham_5(2)$
Es werde $v = (0,0,0,1,1,1)$ empfangen. Dann ist $s = (3,4)$, also ist s das 3-Fache der 5. Spalte von M. Wir korrigieren also v zu $c = (0,0,0,1,1-3,1) = (0,0,0,1,3,1) \in Ham_5(2)$.

3.2.4 Komplexität: Simplexdecodierung

Zur Simplexdecodierung legt man sich am besten zusätzlich zur Generatormatrix M des Simplexcodes eine Liste aller von 0 verschiedenen Vielfachen der Spaltenvektoren von M an, da man diese ja für das Decodierverfahren braucht. Dies geschieht einmal und kann dann immer wieder verwendet werden. Bei binären Codes reicht natürlich die Matrix M selbst. Zur Berechnung des Syndroms s von v benötigt man $2kn$ elementare Operationen. Der Vergleich zwischen s und allen von 0 verschiedenen Vielfachen der Spalten von M erfordert $(q^k - 1)k$ elementare Operationen. Letztlich benötigt die Korrektur von v nur noch eine elementare Operation. Die Komplexität des Algorithmus ergibt sich also zu $\sim kq^k$. Vergleicht man dieses Ergebnis mit der Komplexität der Syndromdecodierung, so erlebt man eine Überraschung. Für unsere Beispiele schien die Simplexdecodierung viel einfacher und übersichtlicher. Algorithmisch stimmen aber beide Komplexitäten überein. Man beachte dabei, dass die Dimension von Hamming-Codes gleich $n - k$ ist.

3.2.5 Anwendung: ECC-Arbeitsspeicher in Computern

Als **ECC-Memory** (**Error-Correcting-Code Memory**) bezeichnet man die Ausprägung von **Arbeitsspeichern** in Computern, bei denen die meisten der üblichen internen Datenfehler entdeckt und korrigiert werden können. ECC wird besonders dort eingesetzt, wo fehlerhafte Daten auf gar keinen Fall toleriert werden können, z. B. bei wissenschaftlichen Rechnern oder Rechnern im Finanzwesen. Da Bitfehler im Arbeitsspeicher von Compu-

tern extrem selten sind, werden dort verbreitet binäre Hamming-Codes eingesetzt. Man ist nämlich hier viel mehr an schneller Zugriffszeit und daher an geringer Redundanz interessiert. Damit schließt sich sozusagen wieder der Kreis zurück zu Hammings Lochkarten.

Man setzt jedoch in erster Linie erweiterte binäre Hamming-Codes ein. Der Grund ist der folgende: Wie wir wissen, kann man bei Hamming-Codes wegen ihres Minimalabstandes $d = 3$ genau einen Fehler korrigieren. Man kann jedoch bei bis zu zwei Fehlern nicht unterscheiden, ob es sich wirklich um genau einen oder etwa doch um zwei Fehler gehandelt hat. Die erweiterten binären Hamming-Codes haben dagegen bei nur einem weiteren Paritätsbit schon Minimalabstand $d = 4$. Hier kann man somit bei einem empfangenen Wort v mit maximal zwei Fehlern unterscheiden, ob genau ein oder genau zwei Fehler aufgetreten sind. Der Decodieralgorithmus ist also mächtiger, er entscheidet nämlich zuerst, ob es sich um einen oder mindestens zwei Fehler handelt. Im ersten Fall korrigiert er den Fehler und im zweiten Fall markiert er das Wort nur als fehlerhaft.

In der Regel eingesetzt werden bei den ECC-Arbeitsspeichern die Erweiterungen einer verkürzten Form des binären Hamming-Codes $Ham_2(6)$ und $Ham_2(7)$, die die Parameter $[39, 32, 4]_2$ bzw. $[72, 64, 4]_2$ haben. Das ECC-Verfahren benötigt daher bei 32-Bit-Rechnerarchitektur 7 Paritätsbits und bei 64-Bit-Architektur 8 Paritätsbits, wobei die Codierung in systematischer Form erfolgt. Bei Codeverkürzungen werden aus einem vorgegebenen Code eine oder mehrere Positionen *weggeschnitten*. Wir werden darauf in Abschn. 5.4 genauer eingehen und die beiden Codes auch explizit konstruieren.

3.2.6 Anwendung: Satellitennavigation mit GPS (Teil 1)

Als zweites Beispiel wollen wir auf Satellitennavigation (**GNSS, Global Navigation Satellite System**) eingehen. Ein **GNSS** basiert auf mehreren Satelliten, die mit Radiosignalen ständig ihre aktuelle Position und die genaue Uhrzeit (sowie andere technische Nutzdaten) ausstrahlen. Aus den Signallaufzeiten von vier Satelliten können spezielle Empfänger dann ihre eigene Position (und Geschwindigkeit) berechnen. Es gibt mehrere Systeme weltweit, insbesondere **GPS** (**Global Positioning System**) der USA, **Galileo** (EU), **GLONASS** (Russland) und **Beidou** (China). GPS stammt aus den späten 1980er-Jahren und war ursprünglich eine Entwicklung für die Navigation der US-Marine (NAVSTAR GPS). Es ist aber heutzutage zumindest teilweise zivil verfügbar und auf deutschen Straßen de facto Standard.

GPS sendet auf fünf verschiedenen Frequenzbändern L1 bis L5, bei denen die Daten-Bits auf die elektromagnetische Welle durch Amplituden- bzw. Phasenmodulation *aufgepropft* werden. Die für zivile Zwecke nutzbaren Bereiche sind – in der Reihenfolge ihrer Einführung – L1 C/A, L2C, L5 und L1C. Wir beschreiben hier zunächst die Codierung bei der ältesten Technik **L1 C/A**. Wie in Abb. 3.1 zu sehen ist, besteht ein Frame von L1 C/A aus 1500 Bits, unterteilt in 5 Subframes mit jeweils 300 Bits und diese bestehen jeweils wieder aus 10 Codeworten mit jeweils 30 Bits. Auf die konkreten Dateninhalte wollen hier jedoch nicht weiter eingehen.

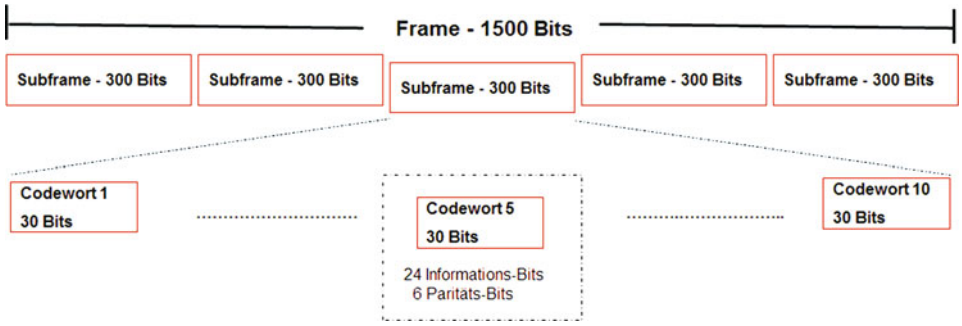


Abb. 3.1 Framestruktur bei GPS L1 C/A (auf Basis [Gar])

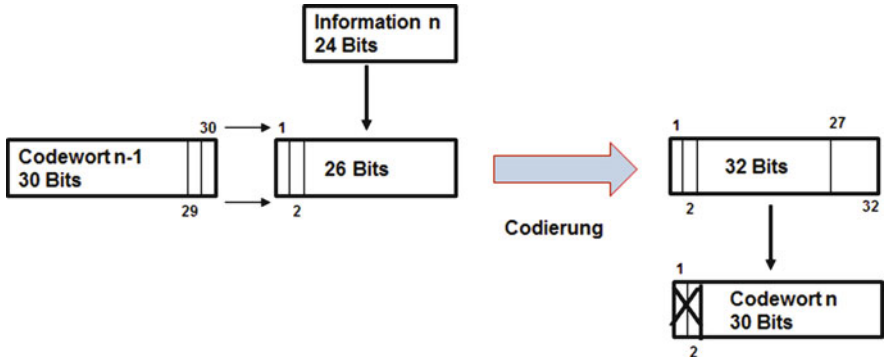


Abb. 3.2 Hamming-Codierung von GPS L1 C/A (auf Basis [Gar])

Jedes Codewort besteht wiederum aus 24 Informationsbits und 6 Paritätsbits. Wie würde man – vor dem Hintergrund des vorangegangenen Beispiels – die Codierung mittels Hamming-Codes naheliegenderweise wählen? Nun, vermutlich würde man sich die Erweiterung einer verkürzten Form des binären Hamming-Codes $Ham_2(5)$ mit Parameter $[30, 24, 4]_2$ hernehmen, denn dieser passt ja genau zu unserer Konstellation. Wir werden in Abschn. 5.4 diese Codeverkürzung aus dem Hamming-Code ableiten. Aber so macht man es bei der Implementierung von L1 C/A nicht. Stattdessen nimmt man direkt den erweiterten Hamming-Code $Ham_2(5)^\wedge$ mit Parameter $[32, 26, 4]_2$. Um aber beim n. Codewort eine Informationsfolge von 26 Bits codieren zu können, nimmt man zu den 24 Informationsbits des n. Worts noch die letzten beiden Bits des vorangegangenen $(n-1)$. Codeworts hinzu. Hierauf wendet man den erweiterten Hamming-Code an, der ein Codewort der Länge 32 in systematischer Form erzeugt. Von diesem lässt man, bevor man es sendet, kurzerhand die ersten beiden Positionen wieder weg. Das Vorgehen ist in Abb. 3.2 visualisiert.

Dieses Vorgehen ist ein kleiner Tabubruch bei Blockcodes, bei denen ja üblicherweise jeder Block separat codiert wird. Hier hängt aber die Codierung des n. Worts von zwei

Bits des vorangegangenen Codeworts ab. Solche Abhängigkeiten werden uns erst wieder in Kap. 9 im Zusammenhang mit Faltungscodes begegnen.

Beim Empfang erfolgt keine automatische Fehlerkorrektur, stattdessen wird nur auf Fehler überprüft, wobei man jeweils die Abhängigkeit vom vorangegangenen Wort berücksichtigen muss. Im Fehlerfall interpoliert man oder ignoriert einfach. Wegen der fehlenden Fehlerkorrektur sendet man jedoch einzelne besonders wichtige Steuerwerte (Zeitdaten, Position des Satelliten etc.) im Frame mehrfach.

Wir kommen auf die anderen Frequenzbänder in den Abschn. 8.1 und 9.6 zurück.

3.3 Golay-Codes und perfekte Codes

Man mag jetzt erwarten, dass wir nach der Familie der Hamming-Codes nun die Familie der Golay-Codes kennenlernen werden. Aber weit gefehlt. Dennoch beginnen wir mit der Konstruktion weiterer Golay-Codes.

3.3.1 Beispiel: Binäre Golay-Codes

Den ternären Golay-Code G_{11} mit Parameter $[12, 6, 5]_3$ samt seiner selbstdualen Erweiterung G_{12} mit Parameter $[12, 6, 6]_3$ haben wir in den Abschn. 2.2, 2.3 detailliert behandelt. Wir kommen nun zur Konstruktion des binären Golay-Codes G_{23} und seiner Erweiterung G_{24}. Die Vorgehensweise ist recht trickreich.

Der erweiterte binäre Golay-Code G_{24} mit Parameter $[24, 12, 8]_2$
Wir betrachten zunächst den erweiterten binären Hamming-Code $Ham_2(3)^\wedge$, wie wir ihn in Abschn. 2.3 beschrieben haben. Wir nennen seine Generatormatrix nun G_1 und den Code C_1, also

$$G_1 = \begin{pmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \end{pmatrix}.$$

Außerdem betrachten wir den zu C_1 äquivalenten Code C_2, der durch Umkehrung der Reihenfolge der ersten sieben Positionen aus C_1 hervorgeht, wobei die Paritätsposition 8 beibehalten wird, also $(c_1, c_2, \ldots, c_7, c_8) \to (c_7, c_6, \ldots, c_1, c_8)$. Dann ist C_2 wie auch C_1 ein selbstdualer $[8, 4, 4]_2$-Code, aber mit Generatormatrix

$$G_2 = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

Es gilt $C_1 \cap C_2 = \{(0,\ldots,0),(1,\ldots,1)\}$, wobei wir diese Rechnung dem Leser überlassen. Man muss dazu einen Vektor der Schnittmenge als Linearkombination sowohl der Zeilen von G_1 als auch der von G_2 ansetzen und das entsprechende lineare Gleichungssystem lösen. Mit diesen Vorbereitungen können wir nun den gesuchten Code G_{24} definieren.

$$G_{24} = \{(v_1 + v_2, w_1 + v_2, v_1 + w_1 + v_2) | v_1, w_1 \in C_1, v_2 \in C_2\} \subseteq (\mathbb{Z}_2)^{24}$$

Seien g_1, g_2, g_3 und g_4 die Zeilen von G_1 (d. h. eine Basis von C_1) und g'_1, g'_2, g'_3 und g'_4 entsprechend von G_2, so überlegt man sich, dass dann die Vektoren $(g_i, 0, g_i)$, $(0, g_i, g_i)$, (g'_i, g'_i, g'_i) für $i = 1, 2, 3, 4$ eine Basis von G_{24} bilden, deren paarweises Skalarprodukt jeweils gleich 0 ist. Somit ist G_{24} ein selbstdualer Code der Länge 24 und Dimension 12. Es bleibt das Minimalgewicht von G_{24} zu bestimmen. Man vergewissert sich dazu zunächst, dass alle eben genannten Basisvektoren von G_{24} ein durch 4 teilbares Gewicht haben. Seien $c = (c_1,\ldots,c_{24})$ und $c' = (c'_1,\ldots,c'_{24})$ zwei Codewörter in G_{24}. Wir betrachten den Träger $Tr(c) = \{i | c_i \neq 0\}$. Wie wir aus Abschn. 2.2 wissen, gilt für das Gewicht der Summe $wt(c + c') = wt(c) + wt(c') - 2|Tr(c) \cap Tr(c')|$. Andererseits folgt aus der Selbstdualität von G_{24}, dass $0 = \langle c, c' \rangle = |Tr(c) \cap Tr(c')|1$ gilt, und $|Tr(c) \cap Tr(c')|$ ist somit durch 2 teilbar. Damit ist jedenfalls das Gewicht der Summe zweier Basisvektoren durch 4 teilbar und – das Argument wiederholt angewendet – das Gewicht jedes Codeworts in G_{24} durch 4 teilbar.

Wir schließen jetzt noch aus, dass G_{24} ein Codewort vom Gewicht 4 enthält. Wegen $wt(v + w) = wt(v) + wt(w) - 2|Tr(v) \cap Tr(w)|$ für $v, w \in (\mathbb{Z}_2)^8$ folgt, dass die Komponenten $v_1 + v_2, w_1 + v_2, v_1 + w_1 + v_2$ in einem Codewort $0 \neq c = (v_1 + v_2, w_1 + v_2, v_1 + w_1 + v_2) \in G_{24}$ alle ein gerades Gewicht haben. Sind alle drei Komponenten ungleich 0, so folgt $wt(c) \geq 8$, da $wt(c)$ durch 4 teilbar ist. Sei also mindestens eine Komponente

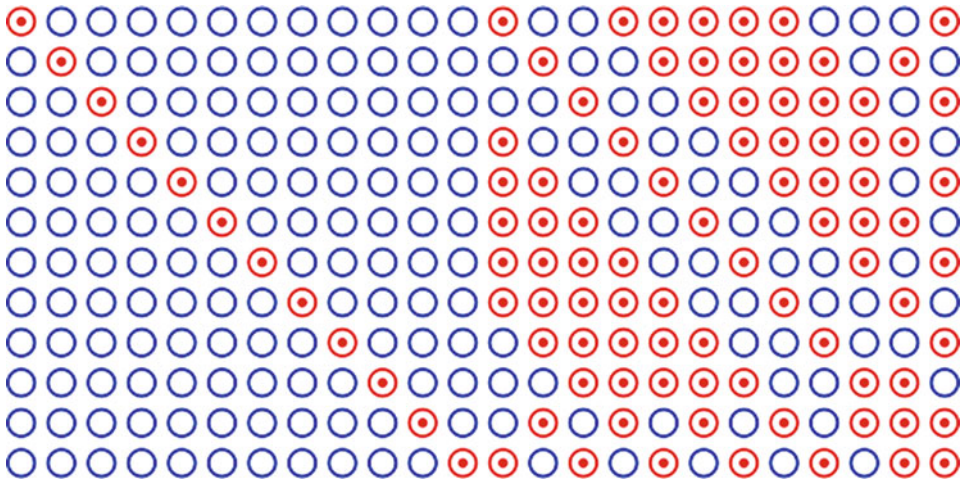


Abb. 3.3 Generatormatrix des erweiterten binären Golay-Codes (Quelle [WPGol])

gleich 0. Wegen $C_1 \cap C_2 = \{(0,\ldots,0),(1,\ldots,1)\}$ folgt dann leicht $v_2 = (0,\ldots,0)$ oder $v_2 = (1,\ldots,1)$ und in beiden Fällen dann auch $wt(c) \geq 8$. Da G_{24} nach Konstruktion Codewörter vom Gewicht 8 enthält, ist schließlich sein Minimalgewicht $d = 8$.

Für all diejenigen, die gern auch mal eine Generatormatrix für den Code G_{24} insgesamt sehen wollen, Abb. 3.3 zeigt eine solche in systematischer Form.

Der binäre Golay-Code G_{23} mit Parameter $[23, 12, 7]_2$
Der Code G_{23} entsteht wiederum durch Streichen der letzten Position in G_{24} und hat damit Länge 23. Man muss sich jetzt noch überlegen, dass die Dimension nach wie vor 12 ist. Dies folgt aber daraus, dass beim Streichen der letzten Position in G_{24} genau die Paritätsziffern der Ausgangscodes C_1 und C_2 im dritten Block von G_{24} gestrichen werden. Letztlich hat nach Streichung der letzten Position etwa der Basisvektor $(g_1, 0, g_1)$ nur noch Gewicht 7.

3.3.2 Kugelpackungsschranke

Weshalb so ein Geheimnis um die Golay-Codes? Hier ist der Schlüssel. Sei dazu K ein endlicher Körper mit $|K| = q$ und C ein $[n,k,d]_q$-Code. Weiterhin sei e die Fehlerkorrekturkapazität von C, also e, maximal mit $d \geq 2e+1$. Wir kommen nochmals auf die Kugeln $B_e(c) = \{v \in K^n | d(v,c) \leq e\}$ vom Radius e um ein Codewort $c \in C$ zurück, wie sie in Abb. 3.4 zu sehen sind.

Diese Kugeln sind disjunkt für verschiedene Codewörter c und c' aus C. Außerdem haben wir uns in Abschn. 1.3 bereits überlegt, wie viele Vektoren $v \in K^n$ eine solche Kugel enthält, nämlich

$$|B_e(c)| = \sum_{i=0}^{e} \binom{n}{i} (q-1)^i .$$

Dabei ist $\binom{n}{i}$ der **Binomialkoeffizient**. Er gibt die Anzahl aller i-elementigen Teilmengen einer n-elementigen Menge an und es gilt $\binom{n}{i} = n!/(i!(n-i)!)$.

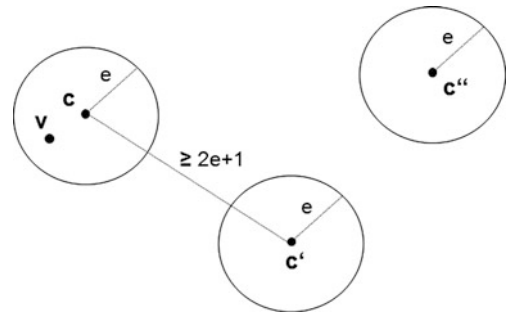


Abb. 3.4 Disjunkte Hamming-Kugeln

Natürlich ist $K^n \supseteq \bigcup_c B_e(c)$, wobei die Vereinigung über alle $c \in C$ gebildet wird. Da die Kugeln um zwei verschiedene Codewörter disjunkt sind, folgern wir zunächst

$$q^n = |K^n| \geq \left|\bigcup_c B_e(c)\right| = \sum_c |B_e(c)| = q^k \sum_{i=0}^{e} \binom{n}{i} (q-1)^i$$

und daraus die

Kugelpackungsschranke (oder **Hamming-Schranke**) $q^{n-k} \geq \sum_{i=0}^{e} \binom{n}{i}(q-1)^i$.

3.3.3 Perfekte Codes

Die Überlegung ist nun die: Es wäre doch *perfekt*, wenn es Codes gäbe, die K^n mit Kugeln vom Radius e genau disjunkt überdecken würden, d. h. wenn in der Kugelpackungsschranke das Gleichheitszeichen gelten würde. Dann nämlich könnte man jedes empfangene Wort v eindeutig in das *eine* Codewort c mit Abstand $d(v,c) \leq e$ korrigieren.

Codes, für die in der Kugelpackungsschranke die Gleichheit gilt, nennt man **perfekt**.

Doch die Euphorie ist etwas verfrüht. Perfekte Codes sind sehr selten, es gilt nämlich der **Klassifikationssatz** von **van Lint** (1971) und **Tietäväinen** (1973).

Sei C ein perfekter linearer Code über dem endlichen Körper K der Länge n mit $0 \subset C \subset K^n$. Dann ist

- C ist ein Hamming-Code $Ham_q(k)$ oder
- C ist der binäre Golay-Code G_{23} oder der ternäre Golay-Code G_{11} oder
- $C = \{(0,\ldots,0),(1,\ldots,1)\}$ ist ein binärer Wiederholungscode von ungerader Länge n.

Von **Jacobus van Lint** (1932–2004) stammt auch der *Klassiker* aller Lehrbücher zur Codierungstheorie [vLi82]. Er war lange Jahre Professor in Eindhoven und trug maßgeblich dazu bei, dass die Codierungstheorie als mathematische Disziplin *hoffähig* wurde.

Für Praxisanwendungen spielt der Begriff **perfekt** und damit die Aussage dieses Satzes keine entscheidende Rolle. Viel interessanter ist die Tatsache, dass die beiden Golay-Codes sozusagen als Ausnahmefälle von perfekten Codes erscheinen. Dies erklärt die Tatsache, dass es keine unendliche Familie von Golay-Codes gibt.

3.3.4 Hamming- und Golay-Codes sind perfekt

Wir wollen uns aber wenigstens davon überzeugen, dass alle Hamming-Codes $Ham_q(k)$ und die beiden Golay-Codes G_{11} und G_{23} wirklich perfekt sind. Dazu müssen wir jeweils die Gleichheit in der Kugelpackungsschranke nachrechnen.

Perfekte Hamming-Codes $Ham_q(k)$
Wie wir wissen, hat $Ham_q(k)$ die Parameter $[n, n-k, 3]_q$ mit $n = (q^k - 1)/(q-1)$ und folglich $e = 1$. Die Summe auf der rechten Seite in der Kugelpackungsschranke vereinfacht sich daher auf zwei Summanden. Wegen $\binom{n}{0} = 1$ und $\binom{n}{1} = n$ ist sie gleich $1 + n(q-1) = 1 + (q^k - 1)/(q-1)(q-1) = q^k$. Wegen $\dim(Ham_q(k)) = n - k$ ergibt sich für die linke Seite $q^{n-(n-k)} = q^k$ und $Ham_q(k)$ ist perfekt.

Perfekte Golay-Codes G_{11} und G_{23}
Wir nutzen die Parameter der Golay-Codes, nämlich $[11, 6, 5]_3$ für G_{11} und $[23, 12, 7]_2$ für G_{23}, und betrachten wieder die Kugelpackungsschranke. Im Fall G_{11} ist $e = 2$ und $q = 3$ und die Summe auf der rechten Seite vereinfacht sich folglich zu $1 + 11 \cdot 2 + 11 \cdot 5 \cdot 4 = 243 = 3^5 = 3^{11-6}$. Dies ist gleich der linken Seite der Kugelpackungsschranke und G_{11} ist damit perfekt. Im Fall G_{23} ist $e = 3$ und $q = 2$. Damit ergibt die Summe auf der rechten Seite $1 + 23 + 23 \cdot 11 + 23 \cdot 11 \cdot 7 = 2048 = 2^{11} = 2^{23-12}$ und G_{23} ist ebenfalls perfekt.

3.3.5 Güte der Golay-Codes

Die perfekten Golay-Codes G_{11} und G_{23} haben eine passable Fehlerkorrekturkapazität von 2 bzw. 3 sowie eine Rate von immerhin $> 1/2$. Sie sind daher auch heute noch vereinzelt im Einsatz. Die erweiterten Golay-Codes G_{12} und G_{24} haben den ein oder anderen Vorteil gegenüber den perfekten Golay-Codes.

- Sie sind selbstdual, d. h. die Generatormatrix ist gleich der Kontrollmatrix.
- Sie haben nur Codewörter, deren Gewicht durch 3 bzw. 4 teilbar ist.
- Treten bei G_{24} maximal vier Fehler auf, so kann man entscheiden, ob es sich um vier Fehler oder nur um bis zu drei Fehler handelt – was G_{23} nicht kann. Im letzteren Fall ist dann wieder die eindeutige Korrektur möglich.

3.4 Wissenswertes und Kurioses rund um die Golay-Codes

Golay-Codes haben eine erstaunliche Faszination in vielen Bereichen von Wissenschaft und Gesellschaft ausgelöst. Dieser Abschnitt soll einen Eindruck davon vermitteln.

3.4.1 Rund um die Golay-Codes

Golay im Original
Die Originalpublikation [Gol] von **Marcel Golay** aus dem Jahr 1949 besteht aus einer guten halben Seite. **Elwyn Berlekamp**, dem wir in Abschn. 7.5 noch begegnen werden, hat die Arbeit von Golay als „best single published page in coding theory" bezeichnet. Die Originalarbeit von Golay ist wenig systematisch. Motiviert von Hammings perfekten Codes sagt er (Zitat): „A limited search has revealed two such cases" und gibt Generatormatrizen für G_{23} und G_{11} an.

Fußballmagazin *Veikkaaja*
Noch eine Anekdote in diesem Zusammenhang: Der ternäre Golay-Code wurde eigentlich zuerst von dem finnischen Fußballfan **Juhani Virtakallio** im Jahre 1947 im Fußballmagazin Veikkaaja veröffentlicht. Aber darin vermutete niemand *in der wissenschaftlichen Welt* ein solches Thema. Warum Virtakallio sich darum gekümmert hat? Gleich weiter unten folgt die Auflösung.

Konstruktionsverfahren für Golay-Codes
Hier noch ein weiteres Konstruktionsverfahren für G_{24}, dem ein kombinatorischer Ansatz zugrunde liegt. Man fasst dabei binäre 24-Tupel als binäre Zahlen auf, also z. B. $(0, \ldots, 0, 1, 1)$ wird identifiziert mit der binären Zahl 11. Die Konstruktion beginnt mit $c_1 = (0, \ldots, 0)$. Dann konstruiert man $c_2, \ldots, c_{12}$ rekursiv gemäß der Regel, dass c_i die kleinste binäre Zahl ist, die sich von allen Linearkombinationen der vorhergehenden c_j an mindestens acht Stellen unterscheidet. Also ist $c_2 = (0, \ldots, 0, 1, \ldots, 1)$ mit genau 16 Nullen und 8 Einsen. Die Bestimmung der weiteren c_i überlassen wir als eine anspruchsvolle Übung dem Leser. G_{24} schließlich wird von den so konstruierten Vektoren $c_1, \ldots, c_{12}$ erzeugt.

Es gibt noch einige andere Möglichkeiten, Golay-Codes zu konstruieren. Eine weitere – nämlich G_{11} und G_{23} als zyklische Codes – werden wir in Abschn. 6.1 besprechen. Die Konstruktion von G_{24} mittels kleiner binärer Hamming-Codes, wie wir sie im letzten Abschnitt dargestellt haben, stammt von **Richard Turyn** (Air Force Cambridge Research Laboratories) aus dem Jahr 1967. Wir haben diese Methode deshalb ausgewählt, weil sie eine schöne Übung im Umgang mit erweiterten und dualen Codes ist.

Endliche einfache Gruppen

Golay-Codes haben als *sporadische* perfekte Codes viele Querverbindungen zu anderen Teilen der Mathematik. Die bekannteste ist vermutlich die Tatsache, dass die – hier nicht genauer definierte – **Automorphismengruppe** der Golay-Codes die berühmten einfachen **Mathieu-Gruppen** M_{11}, M_{12}, M_{23} und M_{24} sind. Diese und noch eine weitere M_{22} wurden von **Emile Mathieu** bereits 1862 und 1873 entdeckt. *Einfach* bedeutet in diesem Zusammenhang, dass die Gruppe nicht weiter *zerlegt* werden kann. In der Klassifikation der endlichen einfachen Gruppen – einem der großen Ergebnisse der Mathematik des ausgehenden 20. Jahrhunderts – sind die Mathieu-Gruppen 5 von insgesamt 26 sporadischen Ausnahmegruppen.

3.4.2 Anwendung: Fußballwette mit ternären Golay- und Hamming-Codes

Wir wollen nun mit unseren ternären Codes bei ODDSET den *totsicheren* Tipp wagen. Dabei beschränken wir uns auf Siegwetten im Fußball, wie das früher auch bei Toto möglich war. ODDSET bietet mittlerweile viel umfangreichere Wettmöglichkeiten, aber wir versuchen es erst mal einfach. Hier ist die Konvention.

1 Sieg Heimmannschaft
0 Unentschieden
2 Niederlage Heimmannschaft = Sieg Auswärtsmannschaft

Man kann sich dabei aus den angebotenen Spielen einige auswählen, vorzugsweise diejenigen, bei denen man einen *guten Riecher* zu haben glaubt. Bei jedem Spiel ist eine Quote angegeben, für Sieg, Unentschieden und Niederlage, je nach Einschätzung des Favoriten und der Ergebnisse der Vergangenheit. Tab. 3.1 zeigt ein fiktives Beispiel.

Wir spielen die Spielart **Einzelwette**, bei der also jeder Einzeltipp eine eigene Wette ist. Dafür muss man seinen Einsatz wählen, mindestens 0,10 € pro Einzelwette. Liegt man also bei einem oder mehreren der Einzeltipps richtig, so berechnet sich der Gewinn je Tipp als Einsatz x Quote. Und los geht's!

Tab. 3.1 Fiktives Beispiel für ODDSET-Wetten mit Quoten für Sieg, Unentschieden und Niederlage

Spiel-Nr.	Heim	Gast	1	0	2
1	Borussia Dortmund	Werder Bremen	2,00	3,30	3,75
2	Borussia Mönchengladbach	Bayer Leverkusen	2,30	3,00	2,80
3	Eintracht Frankfurt	Mainz 05	2,55	3,10	2,80
4	FC Augsburg	Hertha BSC	1,50	3,50	2,00
5	Hamburger SV	1. FC Köln	2,45	2,90	2,65
6	Schalke 04	Bayern München	2,30	2,10	1,50

Wir spielen zunächst $Ham_3(2)$ mit Parameter $[4, 2, 3]_3$. Was soll das heißen? Wir wählen uns vier attraktive Spiele aus und tippen stur nach den Codewörtern des Codes. Das Codewort $(0, 1, 1, -1) = (0, 1, 1, 2)$ beispielsweise ergibt 0 für Spiel 1, 1 für die Spiele 2 und 3 und 2 für Spiel 4. Wir müssen dafür also insgesamt $|Ham_3(2)| = 3^2 = 9$ Tippzettel ausfüllen. Vielleicht bietet ODDSET dafür aber auch eine Art kompakten Systemschein an?! Warum haben wir gerade $Ham_3(2)$ getippt? Da $Ham_3(2)$ ein perfekter Code mit Minimalabstand 3 ist, hat jeder mögliche Ausgang der vier Spiele Hamming-Abstand höchstens 1 von einer unseren Wetten, d. h. wir haben mindestens einmal drei Richtige auf einem Wettschein. Was bringt das? Nehmen wir der Einfachheit halber an, dass wir als Einsatz 1 € pro Einzelwette gesetzt haben und dass die Quote im Schnitt 2,5 betrug, so haben wir für je vier Einzelwetten auf neun Tippzetteln 36 € bezahlt und sicher 7,50 € gewonnen. Wahrscheinlich haben wir auch noch den ein oder anderen richtigen Einzeltipp auf einem anderen Wettschein, dann könnte sich das Ganze vielleicht noch gerade so rechnen.

Versuchen wir es mal etwas mutiger. Wir wenden dieselbe Technik an, nur jetzt mit elf Einzelwetten und dem Golay-Code G_{11} mit Parametern $[11, 6, 5]_3$. Hier müssen wir also $3^6 = 729$ Tipp-Zettel abgeben, wegen $d = 5$ wissen wir aber, dass mindestens einmal neun richtige Tipps dabei sind. Jetzt wird die Ausbeute allerdings katastrophal. Auf 729 Tippzetteln haben wir je elf Einzelwetten abgegeben, also stehen unserem Einsatz von 8019 € gerade mal $9 \cdot 2{,}5 = 22{,}50$ € sicherer Gewinn gegenüber.

Bei der alten 11er-Wette im Toto wären neun Richtige gerade noch so in den Gewinnrängen gewesen, aber das Verhältnis von Einsatz und Ausbeute wäre ähnlich schlecht gewesen. Der Leser ist angehalten, das Ganze noch auf Basis des Hamming-Codes $Ham_3(3)$ mit Parameter $[13, 10, 3]_3$ durchzurechnen.

Dennoch ist bei unserem Vorgehen noch etwas Spielraum. Zum Beispiel lässt sich der Aufwand stark verringern, wenn man eine gewisse Kenntnis der Spiele mitbringt. So gibt es bestimmte Spiele, bei denen eine Mannschaft klarer Favorit ist. Nimmt man diesen Tipp als gegeben an, so kann man mit weit weniger Codewörtern auskommen. Offensichtlich verliert man dadurch die Garantie, dass zumindest drei, neun oder – im Fall $Ham_3(3)$ – zwölf richtige Tipps darunter sind, aber bei den vermeintlich klaren Favoriten erhöht man dabei seine Gewinnchance. Die erwähnten ternären Codes sowie auch ihre erweiterten Versionen sind nach wie vor Grundlage für viele Wettsysteme. Viel Erfolg beim *Austüfteln* – sozusagen in den Fußstapfen unseres bereits oben genannten Fußballfans Virtakallio.

3.4.3 Anwendung: Voyager-Sonden – Jupiter und Saturn

Wir wollen jetzt zum ersten Mal von einer Mission der **NASA** berichten. **NASA** steht für **National Aeronautics and Space Administration** und ist die 1958 gegründete zivile US-Bundesbehörde für Raumfahrt und Flugwissenschaft mit Sitz in Washington, D. C. Der erweiterte binäre Golay-Code G_{24} wurde von der NASA bei den **Voyager-1-** und **-2-**

Abb. 3.5 Der Planet Saturn, aufgenommen von Voyager 1 (Quelle NASA [WPVo1])

Missionen (1977–1981) eingesetzt, um Farbbilder der Planeten Jupiter und Saturn codiert zur Erde zu senden. Abb. 3.5 zeigt eine Aufnahme des Saturn.

G_{24} mit Parameter $[24, 12, 8]_2$ hat eine vernünftige Rate von $1/2$ und kann bis zu drei Fehler innerhalb eines Strings von 24 Bits korrigieren. Bei der Entfernung zwischen Erde und Jupiter bzw. Saturn muss man immerhin von einer Fehlerwahrscheinlichkeit von ca. 0,05 je übertragenem Bit ausgehen. Bei der Bildübertragung wurden jedem Pixel je ein Rot-, ein Grün- und ein Blauwert zwischen 0 und 255 zugeordnet, der die Intensität der jeweiligen Farbe angibt. Weiterhin wurden diese Werte in ihre binäre Darstellung der Länge 8 (1 Byte) umgewandelt und schließlich in einen langen String aneinandergereiht. Dieser Informationsstring von Nullen und Einsen stellt die zu übertragende Information dar. Für das Codieren wurden jeweils zwölf Stellen zu einem Informationsblock zusammengefasst und mittels systematischer Codierung von G_{24} zu einem 24-stelligen Codewort in G_{24} transformiert, welches dann gesendet wurde.

Noch ein Wort zur Mission: Beide Sonden Voyager 1 und 2 waren ursprünglich nur für die Erkundung des Jupiters und Saturn konzipiert. Diese Aufgabe haben sie in den Jahren 1979–1981 mit großem Erfolg erfüllt. Voyager 1 machte sich danach auf in die äußeren Bereiche des Sonnensystems und in den interstellaren Raum, wo sie ca. 2012 eintraf. Sie sendet noch 2016 regelmäßig Daten zur Erde. Bei Voyager 2 entschied man sich aufgrund des großen Erfolgs der Mission, die Bahn der Sonde in Richtung der Planeten Uranus und Neptun umzulenken. Es gab aber mehrere Probleme, an der die Mission hätte scheitern können:

- die extrem geringe Datenrate aufgrund der großen Entfernung,
- die verminderte Energieabgabe der Batterien,
- längere Belichtungszeiten und damit die Gefahr unscharfer Bilder.

Man musste also von der Erde aus die Software massiv überarbeiten, u. a. das zu übertragende Datenvolumen deutlich reduzieren. Dies wurde dadurch erreicht, dass man die Golay-Codierung durch das Reed-Solomon-Verfahren ersetzte. Dadurch jedenfalls konnte man die Rate von $1/2$ bei G_{24} auf $6/7$ erheblich steigern, sogar bei höherer Korrekturkapazität. Wir kommen auf Reed-Solomon-Codes in Kap. 5 ausführlich zu sprechen. Voyager 2 jedenfalls hat seine Mission erfolgreich in den Jahren 1986 bei Uranus und 1989 bei Neptun fortgesetzt und aufsehenerregende Bilder zur Erde gefunkt. Seit der Neptun-Passage befindet sich Voyager 2 wie ihre Schwestersonde Voyager 1 auf dem Weg in die äußeren Bereiche des Sonnensystems und darüber hinaus.

3.4.4 Anwendung: ALE, Automatic Link Establishment

ALE, **Automatic Link Establishment** (automatischer Verbindungsaufbau) ist ein digitales Kommunikationsprotokoll für Kurzwelle zur Etablierung von Sprach- und Datenkommunikation und dafür de facto ein weltweiter Standard. Ziel ist es, eine gewünschte Gegenstation jederzeit erreichen zu können. Nach Verbindungsaufbau kann mit der Kommunikation ggf. in einer anderen Betriebsart, z. B. Mobilfunk, fortgefahren werden Jede eigenständige ALE-Station benötigt geeignete Hard- und Software und besitzt eine eindeutige ALE-Adresse. ALE nutzt G_{24} als FEC-Verfahren (Forward Error Correction). Zusätzlich verwendet ALE aber auch ARQ-Verfahren (Automatic Repeat Request), auf die wir erst in Abschn. 6.5 eingehen werden.

3.4.5 Anwendung: Klinische Diagnostik

Der **fotoakustische Effekt** basiert auf der Erzeugung akustischer Wellen durch die Bestrahlung eines Objekts mit gepulster bzw. modulierter elektromagnetischer Strahlung. Wird elektromagnetische Strahlung von einem Objekt absorbiert, so kommt es zu einer Erwärmung, die wiederum zu einer räumlichen Expansion des Körpers führt. Diese Expansion erzeugt mechanische Wellen, die von einem Schallwandler detektiert werden können, um anschließend als **fotoakustische Bildgebung** visualisiert zu werden. In der Medizin werden Laserquellen im nahinfraroten Bereich zur Anregung genutzt, da in diesem Wellenlängenbereich biologisches Gewebe eine relativ geringe optische Dämpfung aufweist. Auch in diesem Zusammenhang wurden Untersuchungen angestellt mit dem Ziel, durch Codierung der Pulsraten mit perfekten Golay-Codes die Fehlerrate bei der Bildgebung zu vermindern. Weitere Details hierzu findet man in [Mie].

Reed-Muller-Codes

4

Binäre Reed-Muller-Codes Im Jahr 1954 erhielt die Codierungstheorie einen weiteren Schub. **David E. Muller** publizierte die Familie der sog. **Reed-Muller-Codes**. Die Argumentation in der Originalarbeit beruhte auf Boolescher Algebra, wir hingegen nutzen die erst später entwickelte sog. **Plotkin-Konstruktion**, die besser in unseren Ansatz linearer Codes hineinpasst. Was hat aber Mr. Reed mit den Codes zu tun?

Majority-Logic-Algorithmus Nun, **Irving Reed** publizierte im gleichen Jahr ein Decodierverfahren für Reed-Muller-Codes, das aber auch für andere Klassen von Codes anwendbar ist: Die **Majority-Logic-Decodierung**. Das Verfahren funktioniert dann, wenn man im dualen Code genügend viele sog. **orthogonale** Vektoren finden kann. Wir überzeugen uns anhand von Beispielen, dass dies bei Simplex- und Reed-Muller-Codes der Fall ist. Die Majority-Logic-Methode entscheidet per *Mehrheitsvotum*, wo in einem empfangenen Wort ein Fehler aufgetreten ist. Dies funktioniert bei Simplexcodes recht einfach, für Reed-Muller-Code benötigt man allerdings eine **mehrstufige** Variante.

Decodierung von Reed-Muller-Codes Wir nutzen zunächst das 2-stufige Majority-Logic-Verfahren, um anhand eines kleinen Beispiels einen Reed-Muller-Code erster Ordnung *per Hand* zu decodieren. Ein etwas größerer Reed-Muller-Code erster Ordnung – nämlich der der Länge 32 und Dimension 6 – hat große Berühmtheit erlangt. Mit seiner Hilfe haben **Mariner**-Sonden in den Jahren 1969 bis 1972 die damaligen Schwarz-Weiß-Bilder vom **Mars** übertragen. Wie aber hat Houston decodiert? Nicht per Majority-Logic, sondern mit der legendären *festverdrahteten* **Green Machine**.

O. Manz, *Fehlerkorrigierende Codes*, DOI 10.1007/978-3-658-14652-8_4

4.1 Binäre Reed-Muller-Codes

Wir wollen uns in diesem Abschnitt mit einer weiteren Serie von berühmten Codes beschäftigen, den **Reed-Muller-Codes**. Die Namensgeber **Irving Reed** (1923–2012) und **David E. Muller** (1924–2008) waren zwei amerikanische Mathematiker und Informatiker. Im Jahre 1954 hat Muller die Konstruktion der Codes publiziert und Reed hat den zugehörigen Decodieralgorithmus entwickelt. Seine sog. **Majority-Logic-Decodierung** werden wir im nächsten Abschnitt besprechen. Diese beiden Publikationen stellen einen weiteren Meilenstein in der Entwicklung der Codierungstheorie dar. Muller hat seine Codes mittels **Boolescher Algebra** konstruiert, also mit binären Tupeln und den Operatoren UND, ODER und NICHT (Titel der Arbeit „Application of Boolean Algebra to Switching Circuit Design and to Error Detection“ [Mul]). Reed-Muller-Codes werden deshalb in der Regel nur binär über $\mathbb{Z}_2$ definiert, wenngleich sie sich mittels *Gruppenalgebren* auch leicht auf $\mathbb{Z}_p$ verallgemeinern lassen. Es gibt grundsätzlich mehrere Möglichkeiten, sie zu konstruieren. Wir wählen die sog. Plotkin-Konstruktion, die wieder ein Verfahren aufzeigt, aus vorhandenen Codes andere abzuleiten. Allerdings beschränken auch wir uns auf binäre Reed-Muller-Codes. Der Mathematiker **Morris Plotkin** hat die folgende *Aneinanderkettung* von Codes 1960 publiziert.

4.1.1 Plotkin-Konstruktion

Seien C_i zwei lineare $[n, k_i, d_i]_q$-Codes über dem endlichen Körper K mit Generatormatrizen G_i. Der Code $C = C_1 \propto C_2 = \{c = (c_1, c_1 + c_2) | c_i \in C_i\} \subseteq K^{2n}$ heißt **Plotkin-Konstruktion** von C_1 und C_2. Dabei hat C die Generatormatrix

$$G = \begin{pmatrix} G_1 & G_1 \\ 0 & G_2 \end{pmatrix}$$

und ist ein linearer $[2n, k_1 + k_2, d]_q$-Code mit $d = \min\{2d_1, d_2\}$.

Ein ähnliches, aber noch komplizierteres Konstrukt haben wir schon bei der Turyn-Konstruktion des Golay-Codes G_{24} verwendet.

Klar ist jedenfalls, dass C ein linearer Code der Länge $2n$ ist. Offenkundig ist auch, dass für die Zeilenvektoren $b_i^{(1)}$ und $b_j^{(2)}$ von G_1 und G_2, die ja auch Basisvektoren von C_1 und C_2 sind, die Vektoren $(b_i^{(1)}, b_i^{(1)})$ und $(0, b_j^{(2)})$ eine Basis von C bilden für $i = 1, \ldots, k_1$ und $j = 1, \ldots, k_2$. Also ist $\dim(C) = k_1 + k_2$ und G ist eine Generatormatrix von C.

Etwas schwieriger ist wie üblich der Minimalabstand. Dazu betrachten wir wieder den Träger eines Vektors $x = (x_1, \ldots, x_n) \in K^n$, nämlich $Tr(x) = \{i | x_i \neq 0\}$ und $wt(x) = |Tr(x)|$. Aus Abschn. 2.2 wissen wir $wt(x+y) \geq wt(x)+wt(y)-2|Tr(x) \cap Tr(y)|$. Sei nun $0 \neq c = (c_1, c_1 + c_2) \in C$. Damit erhalten wir $wt(c) = wt(c_1) + wt(c_1 + c_2) \geq wt(c_1) + wt(c_1) + wt(c_2) - 2|Tr(c_1) \cap Tr(c_2)| \geq wt(c_2)$, wobei wir bei der letzten Ungleichung

$wt(c_1) \geq |Tr(c_1) \cap Tr(c_2)|$ benutzt haben. Ist $c_2 \neq 0$, dann folgt aus unserer Ungleichung $wt(c) \geq wt(c_2) \geq d_2$. Ist dagegen $c_2 = 0$, so ist $c_1 \neq 0$ und $wt(c) = 2wt(c_1) \geq 2d_1$. Damit haben wir $d \geq \min\{2d_1, d_2\}$ gezeigt. Wählen wir nun speziell c_1 und c_2 mit $wt(c_1) = d_1$ und $wt(c_2) = d_2$, so gilt natürlich $wt((c_1, c_1)) = 2d_1$ und $wt((0, c_2)) = d_2$. Damit ist wirklich $d = \min\{2d_1, d_2\}$.

4.1.2 Binäre Reed-Muller-Codes – ohne Boolesche Algebra

Seien $0 \leq r \leq m$ nichtnegative ganze Zahlen und $n = 2^m$. Wir definieren die binären Reed-Muller-Codes rekursiv mit der Plotkin-Konstruktion. Dabei seien zunächst

- $RM(0, m) = \{(0, \ldots, 0), (1, \ldots, 1)\}$ = binärer Wiederholungscode der Länge 2^m und
- $RM(m, m)$ = alle 2^m-Tupel über $\mathbb{Z}_2$.

Dann ist insbesondere

- $RM(0, 0) = \mathbb{Z}_2^1$,
- $RM(1, 1) = \mathbb{Z}_2^2$,
- $RM(0, 1) = \{(0, 0), (1, 1)\}$.

Für die übrigen Werte von m und r, nämlich $m = 2, 3, \ldots$ und $1 \leq r \leq m$, definieren wir rekursiv $RM(r, m) = RM(r, m-1) \propto RM(r-1, m-1)$ und nennen dies einen **binären Reed-Muller-Code** r. Ordnung.

Die zugehörigen Generatormatrizen $G_{(r,m)}$ berechnen sich gemäß Plotkin-Konstruktion ebenfalls rekursiv gemäß

$$G_{(r,m)} = \begin{pmatrix} G_{(r,m-1)} & G_{(r,m-1)} \\ 0 & G_{(r-1,m-1)} \end{pmatrix}$$

und $RM(r, m)$ hat die Parameter $[2^m, \sum_{i=0}^{r} \binom{m}{i}, 2^{m-r}]_2$.

Das sieht man jetzt recht einfach, da wir die eigentlichen Überlegungen schon bei der Plotkin-Konstruktion angestellt haben. Zunächst muss man aber diese Parameter für die in der obigen Festlegung explizit genannten Codes verifizieren. Beachten muss man dabei allerdings, dass $\sum_{i=0}^{m} \binom{m}{i} = 2^m$ ist.

Wir argumentieren per Induktion, und zwar nach der Summe $r + m$, und verwenden dabei die Parameter in der Plotkin-Konstruktion. Die Länge von $RM(r, m)$ ist demnach

$2 \cdot 2^{m-1} = 2^m$. Für die Dimension gilt

$$\begin{aligned}\dim(RM(r,m)) &= \dim(RM(r,m-1)) + \dim(RM(r-1,m-1)) \\ &= \sum_{i=0}^{r} \binom{m-1}{i} + \sum_{i=0}^{r-1} \binom{m-1}{i} = 1 + \sum_{i=0}^{r-1} \binom{m-1}{i+1} + \sum_{i=0}^{r-1} \binom{m-1}{i} \\ &= 1 + \sum_{i=0}^{r-1} \binom{m}{i+1} = \sum_{i=0}^{r} \binom{m}{i}.\end{aligned}$$

Dabei muss man sich überlegen, dass aufgrund der Definition des Binomialkoeffizienten $\binom{m-1}{i+1} + \binom{m-1}{i} = \binom{m}{i+1}$ gilt. Schließlich ist der Minimalabstand gemäß Plotkin-Konstruktion

$$d(RM(r,m)) = \min\{2 \cdot 2^{m-1-r}, 2^{(m-1)-(r-1)}\} = 2^{m-r}.$$

4.1.3 Beispiel: Binäre Reed-Muller-Codes

Reed-Muller-Codes $RM(r, 1)$

$RM(0,1) = \{(0,0),(1,1)\}$ Parameter $[2,1,2]$
$RM(1,1) = \mathbb{Z}_2^2$ Parameter $[2,2,1]$
$G_{(0,1)} = (1 \quad 1)$
$G_{(1,1)} = E_2 =$ Einheitsmatrix mit zwei Zeilen und Spalten

Reed-Muller-Codes $RM(r, 2)$

$RM(0,2) = \{(0,0,0,0),(1,1,1,1)\}$ Parameter $[2^2,1,2^2]$
$RM(1,2) = RM(1,1) \propto RM(0,1)$ Parameter $[2^2,3,2]$
$RM(2,2) = \mathbb{Z}_2^4$ Parameter $[2^2,4,1]$

$$G_{(0,2)} = \begin{pmatrix} 1 & 1 & 1 & 1 \end{pmatrix}$$

$$G_{(1,2)} = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

$G_{(2,2)} = E_4 =$ Einheitsmatrix mit vier Zeilen und Spalten

Reed-Muller-Codes $RM(r, 3)$

$$RM(0,3) = \{(0,0,0,0,0,0,0,0), (1,1,1,1,1,1,1,1)\} \quad \text{Parameter } [2^3, 1, 2^3]$$
$$RM(1,3) = RM(1,2) \propto RM(0,2) \quad \text{Parameter } [2^3, 4, 2^2]$$
$$RM(2,3) = RM(2,2) \propto RM(1,2) \quad \text{Parameter } [2^3, 7, 2]$$
$$RM(3,3) = \mathbb{Z}_2^8 \quad \text{Parameter } [2^3, 8, 1]$$

$$G_{(0,3)} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

$$G_{(1,3)} = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

$$G_{(2,3)} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

$G_{(3,3)} = E_8 =$ Einheitsmatrix mit acht Zeilen und Spalten

Reed-Muller-Codes $RM(r, 4)$

$$RM(0,4) = \text{ binärer Wiederholungscode} \quad \text{Parameter } [2^4, 1, 2^4]$$
$$RM(1,4) = RM(1,3) \propto RM(0,3) \quad \text{Parameter } [2^4, 5, 2^3]$$
$$RM(2,4) = RM(2,3) \propto RM(1,3) \quad \text{Parameter } [2^4, 11, 2^2]$$
$$RM(3,4) = RM(3,3) \propto RM(2,3) \quad \text{Parameter } [2^4, 15, 2]$$
$$RM(4,4) = \mathbb{Z}_2^{16} \quad \text{Parameter } [2^4, 16, 1]$$

4.1.4 Duale binäre Reed-Muller-Codes

Wie wir ja wissen, macht es immer Sinn, zu einem Code oder einer ganzen Familie von Codes die dualen Codes zu bestimmen. Vielleicht ergeben sich dabei wieder brauchbare neue Codes, in jedem Fall sind duale Codes aber für die Syndromdecodierung wichtig. Bezogen auf die erste Zielsetzung ist hier das Ergebnis eher enttäuschend. Duale Codes von Reed-Muller-Codes sind wieder Reed-Muller-Codes, es gilt nämlich $RM(r,m)^\perp = RM(m-r-1, m)$. Insbesondere ist $RM(r, 2r+1)$ selbstdual.

Wir leiten dieses Ergebnis hier nicht formal her. Wer sich daran versuchen möchte, der argumentiert am besten per Induktion nach m und überlegt mittels $RM(r-1,m) \subseteq$

$RM(r,m)$, dass $RM(m-r-1,m) \subseteq RM(r,m)^{\perp}$ gilt. Die Gleichheit folgt dann wegen $\dim(RM(m-r-1,m)) = 2^m - \dim(RM(r,m))$. Auf jeden Fall sollte man sich aber anhand der obigen Beispiele von der Richtigkeit der Aussage überzeugen.

4.1.5 Güte von binären Reed-Muller-Codes

Für $r = 1$ ergibt sich eine Rate von $(1+m)/2^m$ und ein relativer Minimalabstand von $2^{m-1}/2^m = 1/2$. Asymptotisch geht daher die Rate schnell gegen 0. Für große r ist dieses Verhalten gegenläufig. Bei $r = m-1$ zum Beispiel ergibt sich eine sehr gute Rate von $(2^m-1)/2^m$, die schnell auf 1 zuläuft. Aber der Minimalabstand von 2 ist schon absolut gesehen unbrauchbar. Reed-Muller-Codes sind – was die Anwendung anbetrifft – auch eher von historischem Interesses. Der von Reed zunächst speziell für Reed-Muller-Codes entwickelte Decodieralgorithmus hat allerdings auch darüber hinaus Anwendung gefunden. Auf sein Verfahren gehen wir im nächsten Abschnitt ein.

4.2 Majority-Logic-Algorithmus

Es hat sich bei der Syndromdecodierung linearer Codes C gezeigt, dass es sinnvoll ist, mit der Kontrollmatrix H von C und dem Syndrom eines Vektors zu arbeiten. Mit den Zeilenvektoren z_i von H liegt auch jede Linearkombination $\sum_i a_i z_i$ wieder in $C^{\perp}$ und kann folglich auch für eine **Kontrollgleichung** genutzt werden, d. h. $\langle c, \sum_i a_i z_i \rangle = 0$ für $c \in C$. Der Ansatz bei der **Majority-Logic-Decodierung** ist es nun, sich möglichst *gute* Kontrollgleichungen zu verschaffen. Was *gut* eigentlich heißen soll, müssen wir natürlich gleich noch genau festlegen. Das Verfahren stammt wie bereits gesagt von **Irving Reed** aus dem Jahr 1954 und wurde vorgeschlagen im Zusammenhang mit der Entdeckung der Reed-Muller-Codes durch **David E. Muller**. Für Reed-Muller-Codes gibt es nämlich immer solche *guten* Kontrollgleichungen und Majority-Logic ist dabei schneller als die Syndromdecodierung. Auch beim Majority-Logic-Verfahren beschränken wir uns auf binäre lineare Codes über $\mathbb{Z}_2$.

4.2.1 Orthogonale Vektoren und Kontrollgleichungen

Sei C ein binärer linearer Code der Länge n und $y^{(1)} = (y_1^{(1)}, \ldots, y_n^{(1)}), \ldots, y^{(s)} = (y_1^{(s)}, \ldots, y_n^{(s)})$ Vektoren in $C^{\perp}$. Insbesondere gelten dann für $c \in C$ die Kontrollgleichungen $\langle c, y^{(j)} \rangle = 0$ für $j = 1, \ldots, s$. Sei weiterhin I eine Teilmenge von $\{1, \ldots, n\}$. Man nennt $y^{(1)}, \ldots, y^{(s)}$ **orthogonal bzgl. der Menge I**, wenn alle $y^{(j)}$ an den Positionen $i \in I$ eine 1 haben, aber außerhalb von I mit keinem anderen der $y^{(k)}$ eine 1 gemeinsam haben.

4.2.2 Beispiel: Orthogonale Vektoren

Simplexcode $Sim_2(3)$
Zunächst ein einfaches Beispiel, nämlich der Simplexcode $Sim_2(3)$. Wir wissen aus Abschn. 2.3, dass C folgende Kontrollmatrix hat:

$$H = \begin{pmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}.$$

Dabei sind die

$$\begin{aligned} y^{(1)} &= h_1 &&= (1,1,0,1,0,0,0), \\ y^{(2)} &= h_1 + h_2 + h_3 &&= (1,0,0,0,1,1,0), \\ y^{(3)} &= h_1 + h_2 + h_4 &&= (1,0,1,0,0,0,1) \end{aligned}$$

orthogonal bzgl. der Teilmenge $\{1\}$. Bei einelementigen Teilmengen $\{i\}$ sagt man auch **orthogonal bzgl. der Position i**.

Reed-Muller-Code $RM(1,3)$
Aus dem letzten Abschnitt kennen wir die Generatormatrix des Reed-Muller-Codes $RM(1,3)$ und wissen auch, dass $RM(1,3)$ selbstdual ist. Also ist die Generatormatrix gleichzeitig auch Kontrollmatrix, nämlich

$$H = \begin{pmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix},$$

und die

$$\begin{aligned} y^{(1)} &= h_3 &&= (0,0,1,1,0,0,1,1), \\ y^{(2)} &= h_4 &&= (0,0,0,0,1,1,1,1), \\ y^{(3)} &= h_1 + h_2 + h_3 + h_4 &&= (1,1,0,0,0,0,1,1) \end{aligned}$$

sind orthogonal bzgl. der Menge $\{7,8\}$.

4.2.3 Einfacher Majority-Logic-Algorithmus

Wir können jetzt den **einfachen Majority-Logic-Algorithmus** mit einelementigen Mengen $I = \{i\}$ formulieren. Sei dazu C ein binärer linearer Code der Länge n und

$y^{(1)}, \ldots, y^{(s)}$ orthogonal bzgl. Position i. In einem empfangenen Wort v mögen höchstens $t \leq s/2$ Fehler auftreten. Dann hat v an Position i genau dann einen Fehler, wenn $|\{j \,|\langle v, y^{(j)}\rangle \neq 0\}| > |\{j \,|\langle v, y^{(j)}\rangle = 0\}|$.

Bei der **einfachen Majority-Logic-Decodierung** entscheidet also die Mehrheit der Kontrollwerte $\langle v, y^{(j)}\rangle$, ob an Position i ein Fehler vorliegt oder nicht. Daher auch der Name **Mehrheitslogik**.

- Mehrheit der Kontrollwerte $\neq 0 \Rightarrow$ Fehler an Position i (und damit korrigierbar, da binär)
- Mehrheit bzw. Hälfte der Kontrollwerte $= 0 \Rightarrow$ Position i ist korrekt

Das Problem dabei ist jedoch, dass man zur vollständigen Korrektur orthogonale Vektoren zu *jeder* Position benötigt, dass aber für einen Code nicht immer orthogonale Vektoren existieren müssen.

Wir wollen nun den einfachen Majority-Logic-Algorithmus nachweisen. Um die Bezeichnungen nicht zu unübersichtlich zu machen, nehmen wir $i = 1$ an, d. h. wir wollen entscheiden, ob an Position 1 ein Fehler auftritt oder nicht. Nach Voraussetzung gibt es dann s orthogonale Vektoren $y^{(1)}, \ldots, y^{(s)}$ bzgl. Position 1. Es werde das Codewort $c = (c_1, \ldots, c_n) \in C$ gesendet und das Wort $v = (v_1, \ldots, v_n) \in (\mathbb{Z}_2)^n$ empfangen, wobei v nach Voraussetzung höchstens $t \leq s/2$ Fehler enthält; diese mögen an den Positionen $k_1, \ldots, k_t$ auftreten. Ziel ist es zu entscheiden, ob einer der Fehler an Position 1 des Wortes aufgetreten ist (d. h. $c_1 \neq v_1$) oder ob dies nicht der Fall ist (d. h. $c_1 = v_1$). Jedenfalls wissen wir für $m = 1, \ldots, s$, dass $\langle c, y^{(m)}\rangle = 0$ gilt, können aber andererseits auch $\langle v, y^{(m)}\rangle = v_1 y_1^{(m)} + \ldots + v_n y_n^{(m)}$ berechnen.

Wir betrachten zunächst den Fall, dass $v_1 \neq c_1$ fehlerhaft ist, d. h. $1 = k_1$. Wegen $y_1^{(m)} = 1$ folgt $v_1 y_1^{(m)} \neq c_1 y_1^{(m)}$ für alle $m = 1, \ldots, s$. Wegen der Orthogonalität der $y^{(m)}$ gibt es aber maximal $t - 1$ viele $y^{(m)}$, die an einer der Stellen $k_2, \ldots, k_t$ einen Eintrag 1 haben. Wir betrachten nun die übrigen $y^{(j)}$, wovon es also mindestens $s - (t - 1)$ Stück gibt. Hierbei gilt $c_r y_r^{(j)} = v_r y_r^{(j)}$ für $r = 2, \ldots, n$, da entweder $c_r = v_r$ oder $y_r^{(j)} = 0$ ist. Also folgt für diese j die Aussage $\langle v, y^{(j)}\rangle = \langle v, y^{(j)}\rangle - 0 = \langle v, y^{(j)}\rangle - \langle c, y^{(j)}\rangle = v_1 y_1^{(j)} - c_1 y_1^{(j)} \neq 0$.

Nun untersuchen wir den Fall, dass $v_1 = c_1$ korrekt ist, d. h. $1 \notin \{k_1, \ldots, k_t\}$. Wegen der Orthogonalität der $y^{(m)}$ gibt es nun maximal t viele $y^{(m)}$, die an einer der Stellen $k_1, \ldots, k_t$ einen Eintrag 1 haben. Die übrigen $y^{(j)}$ haben an diesen Stellen Eintrag 0 und es gibt davon mindestens $s-t$ Stück. Für diese j gilt also $v_r y_r^{(j)} = c_r y_r^{(j)}$ für $r = 1, \ldots, n$ und damit $\langle v, y^{(j)}\rangle = \langle c, y^{(j)}\rangle = 0$.

Wegen $t \leq s/2$ haben wir insgesamt soeben gezeigt:

- Ist v_1 fehlerhaft, so ist $\langle v, y^{(j)} \rangle \neq 0$ für wenigstens $s - (t-1) > s/2$ Werte von j.
- Ist v_1 korrekt, so ist $\langle v, y^{(j)} \rangle = 0$ für wenigstens $s - t \geq s/2$ Werte von j.

Also ist v_1 genau dann fehlerhaft, wenn die Mehrheit der $\langle v, y^{(j)} \rangle \neq 0$ ist.

4.2.4 Beispiel: Einfache Majority-Logic-Decodierung

Simplexcode $Sim_2(3)$
Wir führen unser Beispiel $Sim_2(3)$ fort und wollen versuchen, Fehler an der 1. Stelle mit dem einfachen Majority-Logic-Verfahren zu decodieren.

Es werde $v = (0,1,0,1,0,1,0)$ empfangen, wobei maximal ein Fehler aufgetreten sei. Wir berechnen die Kontrollwerte $\langle v, y^{(1)} \rangle = 0$, $\langle v, y^{(2)} \rangle = 1$ und $\langle v, y^{(3)} \rangle = 0$ und schließen daraus, dass die 1. Stelle von v korrekt ist.

Es werde $v = (0,1,1,0,0,1,0)$ empfangen, maximal ein Fehler sei aufgetreten. Wir berechnen wieder die Kontrollwerte $\langle v, y^{(1)} \rangle = 1$, $\langle v, y^{(2)} \rangle = 1$ und $\langle v, y^{(3)} \rangle = 1$, woraus sich ergibt, dass die 1.Stelle von v zu 1 korrigiert werden muss.

Aber Vorsicht: Es werde $c = (0,1,0,1,1,1,0)$ gesendet und $v = (1,1,0,0,0,1,1)$ empfangen. Dabei sind also vier Fehler aufgetreten. Die Kontrollwerte ergeben alle 0, also müsste nach dieser Logik die erste Stelle von v korrekt sein, was offensichtlich ja nicht stimmt. Das ist auch nicht verwunderlich, da der Majority-Logic-Algorithmus eine obere Grenze für die zulässige Anzahl von Fehlern hat, bei deren Überschreitung er auf sinnlose Ergebnisse führen kann.

4.2.5 Mehrstufiger Majority-Logic-Algorithmus

Mit dem einfachen Majority-Logic-Verfahren können wir Reed Muller-Codes leider noch nicht decodieren. Dazu bedarf es der mehrstufigen Variante.

Wir wollen daher jetzt den allgemeinen **Majority-Logic-Algorithmus** formulieren. Sei dazu C ein binärer lineare Code der Länge n und $y^{(1)}, \ldots, y^{(s)}$ orthogonal bzgl. der Menge $I \subseteq \{1, \ldots, n\}$. In einem empfangenen Wort v mögen höchstens $t \leq s/2$ Fehler auftreten. Dann ist die Anzahl der Fehler in v an den Positionen von I genau dann ungerade, wenn $|\{j \,|\langle v, y^{(j)} \rangle \neq 0\}| > |\{j \,|\langle v, y^{(j)} \rangle = 0\}|$.

Auf den Nachweis dieser Verallgemeinerung wollen wir nicht weiter eingehen. Für $I = \{i\}$ ist dies genau der einfache Majority-Logic-Algorithmus, denn „Anzahl der Fehler ungerade“ bedeutet bei nur einer Position eben *Fehler*. Der allgemeine Majority-Logic-Algorithmus hat jedoch offensichtlich einen großen Nachteil, denn bei $|I| > 1$ weiß man nur, dass die Anzahl der Fehler an den Positionen von I gerade oder ungerade ist, aber eben nicht genau, wo die Fehler wirklich stecken. Allerdings kann man das Verfahren

als **mehrstufige Majority-Logic-Decodierung** durchführen, wobei man bei jeder Stufe geeignete Mengen I um ein Element reduziert. Wir wollen das Verfahren in dieser Allgemeinheit nicht weiter präzisieren. Tatsache ist aber, dass es für Reed-Muller-Codes $RM(r, m)$ stets in $r + 1$ Stufen funktioniert.

4.3 Decodierung von Reed-Muller-Codes

4.3.1 Beispiel: 2-stufige Majority-Logic-Decodierung

Reed-Muller-Code $RM(1, 3)$

Die Funktionsweise der mehrstufigen Majority-Logic-Decodierung wollen wir konkret am Beispiel $RM(1, 3)$ erläutern, in diesem Fall ein 2-stufiges Verfahren. Wir wollen wieder durch Mehrheitslogik entscheiden, ob die Position 1 eines empfangenen Worts $v = (v_1, \ldots, v_8)$ fehlerhaft ist oder nicht.

Zunächst schauen wir uns dazu die in Tab. 4.1 dargestellte Liste aller $2^4 = 16$ Codewörter von $RM(1, 3)$ an. Diese ist so zu lesen.

- Unter den Ziffern 1 bis 8 sind die 16 Codewörter von $RM(1, 3)$ aufgeführt.
- Links von den Codewörtern steht jeweils die Regel, wie diese aus der Kontrollmatrix H von $RM(1, 3)$ entstanden sind.

Tab. 4.1 $RM(1, 3)$-Codewörter mit orthogonalen Vektoren

Zeilen H	1	2	3	4	5	6	7	8	1, 2	1, 3	1, 4	1, 5	1, 6	1, 7	1, 8
	0	0	0	0	0	0	0	0							
1	1	0	1	0	1	0	1	0		×		×		×	
2	0	1	0	1	0	1	0	1							
3	0	0	1	1	0	0	1	1							
4	0	0	0	0	1	1	1	1							
$1+2$	1	1	1	1	1	1	1	1							
$1+3$	1	0	0	1	1	0	0	1			×	×			×
$1+4$	1	0	1	0	0	1	0	1		×			×		×
$2+3$	0	1	1	0	0	1	1	0							
$2+4$	0	1	0	1	1	0	1	0							
$3+4$	0	0	1	1	1	1	0	0							
$1+2+3$	1	1	0	0	1	1	0	0	×			×	×		
$1+2+4$	1	1	1	1	0	0	0	0	×	×	×				
$1+3+4$	1	0	0	1	0	1	1	0			×		×	×	
$2+3+4$	0	1	1	0	1	0	0	1							
$1+2+3+4$	1	1	0	0	0	0	1	1	×					×	×

- Rechts von den Codewörtern wird markiert, welche drei Codewörter jeweils zusammen ein System von orthogonalen Vektoren bzgl. der Mengen $I = \{1, i\}$ bilden, für $i = 2, \ldots, 8$.

Wir führen das Verfahren am Beispiel $I = \{1, 2\}$ explizit aus. Dazu schreiben wir uns zunächst für $y^{(1)} = (1, 1, 0, 0, 1, 1, 0, 0)$, $y^{(2)} = (1, 1, 1, 1, 0, 0, 0, 0)$ und $y^{(3)} = (1, 1, 0, 0, 0, 0, 1, 1)$ die Kontrollwerte zu v auf, nämlich $\langle y^{(1)}, v\rangle = v_1 + v_2 + v_5 + v_6$, $\langle y^{(2)}, v\rangle = v_1 + v_2 + v_3 + v_4$ und $\langle y^{(3)}, v\rangle = v_1 + v_2 + v_7 + v_8$. Für ein konkretes v stellen wir anschließend aus der Anzahl der Kontrollwerte $\neq 0$ fest, ob die Anzahl der Fehler in $v = (v_1, \ldots, v_8)$ an den Positionen $\{1, 2\}$ ungerade oder gerade ist. Dies tun wir dann auch sukzessive für alle $\{1, i\}$ und erhalten jeweils die Aussage ungerade oder gerade. Als konkretes Beispiel werde $v = (1, 1, 1, 0, 1, 0, 1, 0)$ empfangen, maximal ein Fehler sei aufgetreten. Die Kontrollwerte für $\{1, 2\}$ sind $1, 1, 1$, also ist die Anzahl der Fehler von v an den Positionen $\{1, 2\}$ ungerade. Für $\{1, 3\}$ sind die Kontrollwerte $0, 0, 1$ und die Anzahl der Fehler an den Positionen $\{1, 3\}$ ist gerade. Man erhält schließlich folgende Tabelle:

$i = 2$	$i = 3$	$i = 4$	$i = 5$	$i = 6$	$i = 7$	$i = 8$
ungerade	gerade	gerade	gerade	gerade	gerade	gerade

Damit muss $v_1 = 1$ korrekt gewesen sein, denn anderenfalls hätte v auch Fehler an den Positionen $3, \ldots, 8$ und damit sieben Fehler.

In unserem Beispiel eines Reed-Muller-Codes 1. Ordnung können wir nun sofort weiter schließen, dass v einen Fehler an Position 2 hat und dass alle anderen Positionen korrekt sind.

4.3.2 Komplexität: Majority-Logic-Decodierung

Obwohl der Majority-Logic-Algorithmus im Zusammenhang mit der Entdeckung der Reed-Muller-Codes entwickelt wurde, funktioniert er auch bei anderen Serien von linearen Codes. Voraussetzung hierzu ist stets die Existenz von orthogonalen Vektoren, die bei Simplex- und Reed-Muller-Codes jedenfalls vorhanden sind. Man kennt auch den eigentlichen Grund hierfür: Die beiden Codes sind nämlich durch *endliche Geometrien* definiert. Auf diese oder ähnliche Art hat man viele weitere Klassen von Codes konstruiert, die dann jeweils mittels des Majority-Logic-Verfahrens decodierbar sind.

Voraussetzung für den Majority-Logic-Algorithmus ist ja, dass die Anzahl der Fehler in einem empfangenen Wort höchstens gleich der Hälfte der Anzahl der orthogonalen Vektoren ist. Ansonsten kann es zu falschen Ergebnissen kommen, wie wir im letzten Abschnitt beim Simplexcode gesehen haben, oder gar zu Widersprüchen, bei denen das Verfahren ohnehin abgebrochen werden muss. Es empfiehlt sich daher, Majority-Logic als BD-Decodierung durchzuführen, die dann nur maximal $s/2$ Fehler je empfangenem Wort korrigiert.

Reed-Muller-Codes $RM(1,m)$ erster Ordnung haben Minimalabstand 2^{m-1} und erlauben daher bei Hamming-Decodierung die Korrektur von maximal $e \leq (2^{m-1}-1)/2 = 2^{m-2} - 1/2$ Fehlern. Interessanterweise gibt es bei $RM(1,m)$ genau $s = (2^m - 2)/2$ orthogonale Vektoren zu jeder 2-elementigen Teilmenge $\{i, j\}$. Mit dem Majority-Logic-Verfahren lassen sich daher maximal $t \leq s/2 = (2^m - 2)/2^2 = 2^{m-2} - 1/2$ Fehler korrigieren. Dies entspricht also genau der Fehlerkorrekturkapazität e des Codes.

Wir wollen uns noch die Komplexität der Majority-Logic-Decodierung für Reed-Muller-Codes $RM(1,m)$ überlegen. Um eine der 2^m Positionen eines empfangenen Worts v korrigieren zu können, benötigen wir $2^m - 1$ Stück an 2-elementigen Teilmengen und dafür jeweils genau $(2^m - 2)/2$ orthogonale Vektoren $y^{(i)}$, die jeweils auch wieder 2^m Positionen haben. Das Berechnen eines Kontrollwertes $\langle v, y^{(i)} \rangle$ benötigt maximal $2 \cdot 2^m$ elementare Operationen. Also ergibt sich für die Berechnung aller Kontrollwerte maximal $2^{(m+1)} \cdot (2^{m-1}-1) \cdot (2^m - 1) \sim 2^{3m}$ elementare Operationen. Dies ist dann auch gleichzeitig die Komplexität des Algorithmus, da man das Zählen der Kontrollwerte ungleich 0 sowie die Korrektur von v selbst vernachlässigen kann.

Die Komplexität der Syndromdecodierung für einen binären linearen Code der Länge n und Dimension k ist $\sim (n-k)2^{(n-k)}$. Für $RM(1,m)$ ergibt sich folglich wegen $n = 2^m$ und $k = m + 1$ eine Komplexität von $\sim 2^{2^m}$ und ist daher asymptotisch viel schlechter als das Majority-Logic-Verfahren.

4.3.3 Anwendung: Mariner-Sonden – Mars

Wir haben uns den berühmtesten Reed-Muller-Code bis zum Schluss aufgehoben, $RM(1,5)$ mit Parameter $[32, 6, 16]_2$. Ihm zu Ehren hier seine komplette Generatormatrix $G_{(1,5)}$:

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

Im Jahr 1969 wurden die beiden Raumsonden **Mariner 6** und **7** von der NASA gestartet. Bei ihrem Vorbeiflug am Mars konnten insgesamt etwa 200 Aufnahmen vom Mars, der Marsoberfläche und den beiden Mars-Monden zur Erde gefunkt werden. Nach einem Fehlstart von Mariner 8 im Jahr 1971 hob dann im selben Jahr **Mariner 9** erfolgreich ab und schwenkte als erste irdische Sonde in die Umlaufbahn um den Mars ein. Bis zu ihrem Betriebsende 1972 machte sie mehrere Tausend Aufnahmen und sendete diese zur Erde. Für die Datenübertragung war der binäre Reed-Muller-Code von Länge

$2^5 = 32$ und mit $2^6 = 64$ Codewörtern implementiert. Die 64 Codewörter entsprachen jeweils dem *Grauwert* (Helligkeit) eines Bildpunktes auf den damals natürlich noch üblichen Schwarz-Weiß-Aufnahmen. Die Kanalcharakteristik, die Bildauflösung und die Aufnahme- und Übertragungszeiten machten einerseits diese Grauwertstufung und anderseits eine Wortlänge von reichlich 30 Bits sinnvoll. Der Code hatte also – nicht zuletzt auch aus technischen Gründen – eine für heutige Verhältnisse miserable Rate von $3/16$. Aufgrund seines Minimalabstandes von $2^4 = 16$ kann der Code wegen $7 \leq (16-1)/2$ bis zu sieben Fehler korrigieren. Auf der Strecke zwischen Mars und Erde musste man nicht zuletzt wegen der damals im Vergleich zu heute rückschrittlichen Übertragungstechnik von einer Bitfehlerwahrscheinlichkeit von 0,05 ausgehen. Durch den Einsatz des Reed-Muller-Codes konnte jedoch die Fehlerrate auf 0,0001 reduziert werden.

4.3.4 Anwendung: Green Machine der NASA

Die 2-stufige Majority-Logic-Decodierung funktioniert natürlich auch für $RM(1,5)$. Trotz alledem: Bei Mariner kam eine *hartverdrahtete* Decodiermethode zum Einsatz, die im NASA-Sprachgebrauch nach ihrem Konstrukteur **Green Machine** genannt wurde. Hier ist eine Basisvariante, die eigentliche Green Machine ist komplizierter, aber auch schneller.

Zuerst werden dabei in allen 64 Codewörtern die Nullen (0) in minus Einsen (-1) umgewandelt und dann mit dem *klassischen* Skalarprodukt „$\cdot$“ über $\mathbb{R}$ gerechnet, auch wenn das bei all der Codierungstheorie eher *archaisch* anmutet. Bei der operativen Decodierung ändert man bei jedem empfangenen Wort v ebenfalls die Nullen (0) in minus Einsen (-1) und multipliziert v mit allen 64 Codewörtern $c \in RM(1,5)$. Sobald ein Ergebnis 16 oder größer dabei herauskommt, ändert man v in dieses Codewort.

Warum funktioniert das? Zunächst muss man wissen, dass alle Codewörter von $RM(1,5)$ außer $(0,\ldots,0)$ vom Gewicht 0 und $(1,\ldots,1)$ vom Gewicht $2^5 = 32$ das Gewicht $2^4 = 16$ haben. Also ist auch der Hamming-Abstand zwischen zwei Codewörtern stets 0, 16 oder 32. Das gilt übrigens allgemein: $RM(1,m)$ hat außer $(0,\ldots,0)$ und $(1,\ldots,1)$ nur Codewörter vom Gewicht $d = 2^{m-1}$.

Es werde also das Wort v empfangen. Dann ist natürlich $v \cdot v = 32$. Wir betrachten zunächst den Fall, dass kein Fehler aufgetreten ist, also v ein Codewort ist. Wir multiplizieren nun (in Gedanken) v mit allen Codewörtern $c \in RM(1,5)$. Ist $d(v,c) = 0$, so ist $v \cdot c = v \cdot v = 32$. Ist $d(v,c) = 32$, so haben v und c keine einzige gemeinsame Koordinate und wegen der Umwandlung von 0 in -1 gilt $v \cdot c = -32$. Im Fall $d(v,c) = 16$ letztlich sind 16 Koordinaten von v und c gleich und die übrigen 16 verschieden, also $v \cdot c = 16 - 16 = 0$. Bei unserem Decodieralgorithmus wird damit v zu c mit $d(v,c) = 0$ *korrigiert*, also unverändert belassen. In diesem Fall liefert der Algorithmus also das richtige Ergebnis.

Tritt in v genau ein Fehler auf, sagen wir an der Position i, dann unterscheidet sich das Skalarprodukt $v \cdot c$ von dem ohne Fehler genau beim i. Summanden, es steht nämlich dort

eine $+1$ anstelle -1 oder umgekehrt. Insbesondere unterscheidet sich das Skalarprodukt $v \cdot c$ insgesamt um $+2$ oder -2.

Bei bis zu sieben Fehlern ergibt sich daher eine Abweichung zwischen -14 und $+14$. Daher hat das Skalarprodukt $v \cdot c$ mit dem richtigen Codewort $c \in RM(1,5)$ mindestens den Wert 18, mit allen falschen aber höchstens den Wert 14. Dies zeigt, dass der Decodieralgorithmus bis zur Fehlerkorrekturkapazität von 7 korrekt funktioniert.

Reed-Solomon-Codes

5

Endliche Körper – reloaded Unser Konzept, das Codieren *algebraisch* anzugehen, hat bis jetzt einige Erfolge gezeigt. Das ermutigt uns, das Ganze noch etwas weiter zu treiben. Wie wär's denn z. B. mit **Polynomen** $f(x)$ über dem endlichen Körper K in einer Unbestimmten x? Damit kann man auch gut rechnen, man kann sie beispielsweise multiplizieren und dividieren – Letzteres möglicherweise mit Rest. Außerdem kann man ihre **Nullstellen** bestimmen und sie an anderen Stellen **auswerten**. Einen ersten Vorteil verschafft uns dieser Ansatz unmittelbar: Die Beschränkung auf sehr kleine Körper K erweist sich nämlich immer öfter als Hindernis. Mittels Polynomen kann man **Körpererweiterungen** bilden, insbesondere Körper mit 2^m Elementen. Besonders wichtig ist der Körper mit $2^8 = 256$ Elementen. Warum? Nun, das **Byte** hat sich hartnäckig als digitale Einheit von 8 Bits gehalten.

Reed-Solomon-Codes und MDS-Codes Den Polynomansatz bringen wir gleich mal zur Anwendung – besser gesagt **Irving Reed** und **Gustave Solomon** haben das 1960 bereits für uns getan. Sie haben Codes definiert, bei denen die n-Tupel aus Polynomauswertungen bestehen – die **Reed-Solomon-Codes** – und deren Verwendung in der Praxis auch heute noch Stand der Technik ist. Bevor wir das Verfahren wie gewohnt an einigen Beispielen einüben, fällt uns im Vorbeigehen auf, dass für Reed-Solomon-Codes in der wichtigen **Singleton-Schranke** das Gleichheitszeichen gilt. Diese sog. **MDS-Codes** (Maximum Distance Separable) weisen bei gegebener Länge die bestmögliche Relation von Rate einerseits und Minimalabstand andererseits auf.

Einige Anwendungen der Reed-Solomon-Codes Nachdem wir gerade die Praxistauglichkeit von Reed-Solomon-Codes besonders hervorgehoben haben, sollten wir sie doch gleich etwas systematischer beleuchten. So wurde die NASA-Sonde **Voyager 2** auf ihrem Weg zu **Uranus** und **Neptun** *remote* von Golay- auf Reed-Solomon-Codes umgestellt. Auch **Galileo** und **Huygens** haben auf ihren Umlaufbahnen um **Jupiter** bzw. **Saturn** Fotos via Reed-Solomon-Codes zur Erde gesandt. Die *allgegenwärtigen* **2-D-Barcodes**

O. Manz, *Fehlerkorrigierende Codes*, DOI 10.1007/978-3-658-14652-8_5

(z. B. QR, Aztec und DataMatrix) korrigieren mögliche Lesefehler mittels Reed-Solomon-Codes. Anwendung finden letztere auch beim **NATO-Militärfunk** MIDS, beim **Digitalfernsehen**, beim schnellen Internet mit **DSL** sowie bei **CDs** und **DVDs**, wie wir gleich genauer studieren werden.

Verkürzte Codes und Cross-Interleaving Auch wenn sich Reed-Solomon-Codes als höchst effektiv erwiesen haben, so passen sie doch nicht immer eins zu eins zur gewünschten Anwendung. Oft ist dabei die Länge nicht ganz adäquat. Dieses Problem löst man, indem man bei einem zu langen Code eine **Verkürzung** vornimmt, also die Länge reduziert, ohne die Güte des Codes wesentlich zu verschlechtern. Zum Beispiel sind Verkürzungen von Reed-Solomon-Codes wieder MDS-Codes. In manch anderen Anwendungen treten die Fehler nicht statistisch gestreut, sondern eher in Bündeln (**Bursts**) auf. Da ist in der Regel auch der beste Code überfordert – auch Reed-Solomon-Codes. Dieses Problem kann man aber dadurch lösen, indem man die Codewörter nicht einfach absendet, sondern sie vorher einem **Interleaving** unterzieht, d. h. man liest sie zunächst zeilenweise in eine Matrix ein und sendet diese Matrix spaltenweise. Man überlegt sich leicht, dass dann die Bursts auf mehrere Codewörter verteilt werden.

Audio-CDs und Daten-DVDs Dies führt uns zum Musterbeispiel von Bursts – nämlich **Kratzer** auf der CD, was insbesondere bei **Audio-CDs** das Hörerlebnis einer alten Langspielplatte wieder in Erinnerung rufen könnte. Aber dem beugt man vor, indem man für die Anwendung adäquate **verkürzte Reed-Solomon-Codes** verwendet, die man beim Brennen der CD noch zusätzlich einem **Interleaving** unterzieht. Beim optischen Abtasten der CD – also beim Musikhören – wird das Interleaving wieder rückgängig gemacht, sodass vorhandene Kratzer von den Reed-Solomon-Codes weitgehend selbstständig korrigiert werden können. Wir gehen das Ganze Schritt für Schritt durch. Das Verfahren funktioniert natürlich auch – leicht modifiziert – bei **Daten-DVDs**.

5.1 Endliche Körper – reloaded

Eigentlich sind wir mit vergleichsweise bescheidenen algebraischen Mitteln bis jetzt ziemlich weit gekommen. Wir haben uns etwas an endliche Körper K gewöhnen müssen anstelle der gewohnten reellen Zahlen $\mathbb{R}$, aber eigentlich sind wir weitgehend mit den binären und ternären Zahlen ausgekommen, sehr selten auch mal $\mathbb{Z}_5$. Wir haben den Vektorraum K^n der n-Tupel über K betrachtet und seine Unterräume mit Basis und Dimension. Genau das waren ja unsere Codes. Wichtig war auch noch das Skalarprodukt $\langle\ ,\ \rangle$, das aber genauso berechnet wird, wie wir es etwa von $\mathbb{R}^3$ her kennen. Um bei unserem Thema noch einen Schritt weiterzukommen, benötigen wir jetzt ein klein wenig mehr Algebra.

5.1.1 Polynome über endlichen Körpern K

Zunächst betrachten wir genau wie über $\mathbb{R}$ die **Polynome** $f(x) = a_n x^n + \ldots + a_1 x + a_0 = \sum_i a_i x^i$ mit einer Variablen x und Koeffizienten in K. Die Menge aller solcher $f(x)$ nennen wir $K[x]$. Als **Grad** von $f(x)$ bezeichnet man den größten Exponenten, mit dem x in $f(x)$ vorkommt. Ist also $a_n \neq 0$, so ist der Grad von $f(x)$ gleich n, kurz $\text{grad}(f(x)) = n$.

Sei k eine natürliche Zahl. Wir bezeichnen mit $K[x]_{k-1}$ die Polynome vom Grad höchstens $k-1$. Dann ist $K[x]_{k-1}$ ein k-dimensionaler Vektorraum über K mit Basis $\{1, x, x^2, \ldots, x^{k-1}\}$. Die lineare Unabhängigkeit ergibt sich hierbei direkt aus dem Koeffizientenvergleich bei Polynomen.

Polynome über K kann man auch mit Rest dividieren. Wenn diese Division aufgeht, hat man einen Teiler gefunden. Hierfür ein kleines Beispiel: Wir nehmen $f(x) = x^3 + 1 \in \mathbb{Z}_2[x]$ und erkennen natürlich sofort, dass $f(x)$ bei 1 eine Nullstelle hat, d. h. $f(1) = 0$. Daher dividieren wir $f(x)$ durch $x+1$, und erhalten

$$
\begin{array}{l}
(x^3+1) : (x+1) = x^2 + x + 1 \\
\underline{x^3 + x^2} \\
\qquad x^2 + 1 \\
\qquad \underline{x^2 + x} \\
\qquad\qquad x + 1 \\
\qquad\qquad \underline{x + 1} \\
\qquad\qquad\quad 0.
\end{array}
$$

Also ist $f(x) = (x+1)(x^2 + x + 1)$. Dieses Verfahren wiederholt angewandt zeigt:

Ein Polynom $f(x)$ vom Grad n hat höchstens n (nicht notwendig verschiedene) **Nullstellen**, d. h. Elemente $\alpha_i \in K$ mit $f(\alpha_i) = 0$. Im Falle von n Nullstellen ergibt sich $f(x) = a_n(x - \alpha_1) \cdots (x - \alpha_n)$ und $f(x)$ lässt sich daher in Polynome vom Grad 1 faktorisieren. Manche Polynome lassen sich auch nur in Faktoren von größerem Grad zerlegen. Polynome, bei denen man keinen echten Teiler vom Grad ≥ 1 findet, nennt man **irreduzibel**. Irreduzible Polynome lassen sich also nicht faktorisieren. Natürlich kann man Polynome auch dividieren, bei denen die Division nicht aufgeht. Dann bleibt eben ein Rest übrig, dessen Grad kleiner als der des Divisors ist (**Division mit Rest**).

Was haben wir nun konkret davon, in der Codierungstheorie zusätzlich zu Vektorräumen auch Polynome zu verwenden? Nun, da gibt es mehrere Gründe.

- Wir haben bislang noch nicht gesehen, wie man endliche Körper konstruiert, die nicht nur p Elemente besitzen (für eine Primzahl p), sondern p^m. Solche benötigen wir nämlich ab jetzt für einige unserer Codes. Körper mit p^m Elementen konstruiert man mit Polynomen, wie wir gleich unten in diesem Abschnitt sehen werden.
- Man kann sich auch überlegen, Codes durch **Auswertungen** von Polynomen an diversen Elementen $\gamma_i \in K$ zu definieren, indem man daraus Tupel geeigneter Länge bildet, also $c = (\ldots, f(\gamma_i), \ldots)$. Wenn man die Nullstellen der verwendeten Polynome $f(x)$ kennt, kann man das möglicherweise so geschickt anstellen, dass viele der $f(\gamma_i) \neq 0$ sind, also das Gewicht $wt(c)$ möglichst groß ist. Dieser Ansatz ist die Grundidee für **Reed-Solomon-Codes**, wie Abschn. 5.2 zeigen wird.
- Ein weiterer Grund, Polynome in unsere Überlegungen einzubeziehen, ist der, dass bei sog. **zyklischen Codes** die Codewörter als Vielfache eines Polynoms interpretiert werden. Mit diesem Ansatz lassen sich Codier- und Decodierverfahren besonders effizient implementieren. Details vertagen wir, bis in Kap. 6 dieses Thema ausführlich besprochen wird.

5.1.2 Konstruktion von Körpern K mit 2^m Elementen

Das folgende Konstruktionsprinzip funktioniert zwar für alle Primzahlenpotenzen p^m, wir werden uns hier aber auf Körper mit 2^m Elementen konzentrieren, da nur diese bei binärer Datenübertragung in der Praxis Anwendung finden. Um also einen Körper mit 2^m Elementen zu konstruieren, nehmen wir uns zunächst den wohlbekannten Grundkörper $\mathbb{Z}_2 = \{0, 1\}$. Nun kommt die eigentliche Herausforderung: Wir benötigen nämlich ein irreduzibles Polynom $f(x) \in \mathbb{Z}_2[x]$ vom Grad $\text{grad}(f(x)) = m$. Solche gibt es immer (was wir hier nicht allgemein nachweisen wollen und *können*), man sieht es ihnen aber leider nicht so einfach an. Falls aber m nicht allzu groß ist, kann man jedoch leicht mittels Division mit Rest für alle Polynome vom Grad $\leq m/2$ überprüfen, ob die Division aufgeht und ob sich $f(x)$ daher echt faktorisieren lässt oder nicht.

Für ein irreduzibles Polynom $f(x)$ vom Grad m konstruiert man den gesuchten Körper K mit 2^m Elementen, indem man alle Polynome in $\mathbb{Z}_2[x]$ modulo $f(x)$ liest, d. h. durch $f(x)$ mit Rest dividiert. Die Reste modulo $f(x)$ erben auf diese Weise die Addition und Multiplikation von $\mathbb{Z}_2[x]$. Formal heißt das

$$K = \{g(x) (\text{mod } f(x)) | g(x) \in \mathbb{Z}_2[x] \text{ und } \text{grad}(g(x)) < m\}.$$

Man muss natürlich zunächst die Körpereigenschaften nachprüfen. Kritisch ist dabei vor allem wieder, dass jeder Rest $g(x) \neq 0$ ein multiplikatives Inverses besitzen muss. Das ist aber genau der Grund, warum $f(x)$ irreduzibel gewählt wird. Dann haben nämlich $g(x)$ und $f(x)$ keinen gemeinsamen Teiler vom Grad ≥ 1 und der Nachweis eines multiplikativen Inversen funktioniert wie beim Körper $\mathbb{Z}_p$, wobei man hier den euklidischen Algorithmus und damit die Vielfachsummendarstellung des größten gemeinsamen Teilers in $\mathbb{Z}_2[x]$ verwenden muss.

Zur Erinnerung: Setzt man $r_0(x) = f(x)$ und $r_1(x) = g(x)$, so ist der euklidische Algorithmus die iterierte Division mit Rest der Form $r_i(x) = r_{i+1}(x)q_{i+2}(x) + r_{i+2}(x)$, wobei $q_i(x), r_i(x) \in \mathbb{Z}_2[x]$ und $\text{grad}(r_{i+2}(x)) < \text{grad}(r_{i+1}(x))$. Der Leser ist eingeladen, den Nachweis eines multiplikativen Inversen formal durchzuführen und dazu eventuell einen Blick in Abschn. 2.1 zu werfen.

Wir setzen zur Abkürzung $t = x \pmod{f(x)}$, dann ist $t \in K$ und $\{t^0 = 1, t, t^2, \ldots, t^{m-1}\}$ ist eine Basis von K als $\mathbb{Z}_2$-Vektorraum, folglich $|K| = |\mathbb{Z}_2|^m = 2^m$.

Wir führen nun einige konkrete Beispiele vor. Zum Eingewöhnen erst mal einen Körper K mit $2^2 = 4$ Elementen. Offenbar ist $f(x) = x^2 + x + 1 \in \mathbb{Z}_2[x]$ irreduzibel. Sei wieder $t = x \pmod{f(x)}$. Dann ist $t^2 = x^2 \pmod{f(x)} = x + 1 \pmod{f(x)} = t + 1$ und dies ergibt für die Multiplikationstafel sämtlicher Elemente $\{0, 1, t, t + 1\}$ von K:

	0	1	t	$t+1$
0	0	0	0	0
1	0	1	t	$t+1$
t	0	t	$t+1$	1
$t+1$	0	$t+1$	1	t

Das Element t ist nicht besser *greifbar*. Man muss also mit solchen Multiplikationsregeln *leben*.

Als nächstes kommen wir zum Körper K mit $2^3 = 8$ Elementen. Das Polynom $f(x) = x^3 + x + 1 \in \mathbb{Z}_2[x]$ ist irreduzibel, da es keine Nullstelle 0 oder 1 hat. Sei wieder $t = x \pmod{f(x)}$. Dann ist $t^3 = t + 1$ und die Multiplikationstafel für K lautet:

	0	1	t	$t+1$	t^2	t^2+1	t^2+t	t^2+t+1
0	0	0	0	0	0	0	0	0
1	0	1	t	$t+1$	t^2	t^2+1	t^2+t	t^2+t+1
t	0	t	t^2	t^2+t	$t+1$	1	t^2+t+1	t^2+1
$t+1$	0	$t+1$	t^2+t	t^2+1	t^2+t+1	t^2	1	t
t^2	0	t^2	$t+1$	t^2+t+1	t^2+t	t	t^2+1	1
t^2+1	0	t^2+1	1	t^2	t	t^2+t+1	$t+1$	t^2+t
t^2+t	0	t^2+t	t^2+t+1	1	t^2+1	$t+1$	t	t^2
t^2+t+1	0	t^2+t+1	t^2+1	t	1	t^2+t	t^2	$t+1$

Für den Körper K mit $2^4 = 16$ Elementen kann man das irreduzible Polynom $f(x) = x^4 + x + 1 \in \mathbb{Z}_2[x]$ verwenden. Für $t = x(\text{mod } f(x))$ gilt diesmal $t^4 = t + 1$ und die recht umfangreiche Multiplikationstafel überlassen wir dem Leser als Übung.

Den Körper K mit $2^5 = 32$ Elementen kann man mit dem irreduziblen Polynom $f(x) = x^5 + x^2 + 1 \in \mathbb{Z}_2[x]$ konstruieren. Für $t = x(\text{mod } f(x))$ gilt nun $t^5 = t^2 + 1$, aber die noch umfangreichere Multiplikationstafel sollten wir jetzt besser sein lassen.

Weil in Anwendungen häufig verwendet, hier noch kurz zum Körper K mit $2^8 = 256$ Elementen. Dass das Polynom $f(x) = x^8 + x^4 + x^3 + x^2 + 1 \in \mathbb{Z}_2[x]$ irreduzibel ist, muss man wieder kontrollieren. Der gesuchte Körper K besteht dann aus allen Resten von Polynomen in $\mathbb{Z}_2$ [x] bei Division durch $f(x)$, also $K = \{g(x)(\text{mod } f(x))|g(x) \in \mathbb{Z}_2[x]$ und $\text{grad}(g(x)) < 8\}$. Die Multiplikationstafel hat jetzt 256 Zeilen und Spalten und wir verzichten natürlich auch hier darauf, sie per Hand auszurechnen. Dies können **Schieberegister** viel schneller und zuverlässiger, wie wir in Abschn. 6.2 genauer sehen werden. Dennoch ein kleines Multiplikationsbeispiel: Wir betrachten t^5 und $t^4 + t + 1 \in K$, wobei wir wieder $t = x(\text{mod } f(x))$ gesetzt haben. Dann ist $t^5(t^4 + t + 1) = t^9 + t^6 + t = t^6 + t^5 + t^4 + t^3$.

Der Körper K mit $2^8 = 256$ Elementen tritt deswegen in Anwendungen häufig auf, weil sich das **Byte** – nämlich als Paket von 8 Bits – nach wie vor standhaft als digitale Einheit hält. Die 8 Bits betrachtet man dann als Koeffizienten eines Polynoms vom Grad maximal 7 und K selbst als Alphabet eines Codes.

Wer auch gerne mal ein kleines Beispiel eines Körpers mit einer anderen Primzahl als 2 sehen will, der kann sich zur Übung an $|K| = 3^2 = 9$ versuchen. Mit dem irreduziblen Polynom $f(x) = x^2 + 1 \in \mathbb{Z}_3[x]$ und $t = x(\text{mod } f(x))$ kann's losgehen, die Multiplikationstafel aufzustellen. Ein kurzer Hinweis zur Vorsicht: Es ist $t^2 = -1$, denn man muss ja bei $\mathbb{Z}_3$ zwischen $+1$ und -1 unterscheiden.

Noch eine Anmerkung zum Schluss: Es gibt in der Regel mehrere irreduzible Polynome $f(x) \in \mathbb{Z}_p[x]$ vom $\text{grad}(f(x)) = m$, mit denen man einen Körper K mit p^m Elementen konstruieren kann. Dabei ergeben sich zwar formal unterschiedliche Multiplikationstafeln, aber alle solche Körper K sind *gleich gut*, man sagt dazu **isomorph**.

5.2 Reed-Solomon-Codes und MDS-Codes

5.2.1 Singleton-Schranke

Wir haben in den vorangegangenen Abschnitten bereits Schranken für Codes kennengelernt – die Gilbert-Schranke und die Kugelpackungsschranke. Schranken geben Relationen zwischen wesentlichen Parametern eines Codes an. Es gibt eine ganze Reihe weiterer solcher Schranken. Wir wollen diesen Abschnitt mit der möglicherweise wichtigsten beginnen.

Sei C ein linearer $[n, k, d]$-Code über K. Dann gilt die sog. **Singleton-Schranke** $d \leq n - k + 1$.

Das lässt sich ganz leicht einsehen. Wir betrachten nämlich die Abbildung $\alpha\colon K^n \to K^{n-d+1}$, welche die letzten $d-1$ Stellen eines n-Tupels abschneidet, also formal $\alpha((x_1, \ldots, x_n)) = (x_1, \ldots, x_{n-d+1})$. Für zwei verschiedene Codewörter c und c' in C müssen dann auch $\alpha(c)$ und $\alpha(c')$ verschieden sein. Sonst nämlich würden sich c und c' nur maximal an den letzten $d-1$ Stellen unterscheiden und es würde $d(c, c') \leq d - 1$ gelten. Also enthält K^{n-d+1} mindestens so viele Elemente wie C. Dies zeigt $k = \dim(C) \leq \dim(K^{n-d+1}) = n - d + 1$.

5.2.2 MDS-Codes

Wie bei der Kugelpackungsschranke und den perfekten Codes benennen wir auch hier eine Klasse von Codes danach, dass in der Singleton-Schranke das Gleichheitszeichen steht.

Ein linearer $[n, k, d]$-Code über K heißt **MDS-Code** (**Maximum Distance Separable**), wenn in der Singleton-Schranke die Gleichheit $d = n - k + 1$ gilt.

Kurz zur Begriffserklärung: Bei MDS-Codes werden je zwei verschiedene Codewörter durch jeweils k ihrer Stellen *getrennt*, d. h. egal welche k Stücke der Koordinaten c_i, c_i' man aus verschiedenen Codewörtern $c = (c_1, \ldots, c_n)$ und $c' = (c_1', \ldots, c_n')$ herausgreift, es können dann nie alle gleich sein.

Die Erklärung hierzu liefert bereits implizit die obige Herleitung der Singleton-Schranke. Für MDS-Codes ist dann nämlich α eine Abbildung von K^n nach K^k und damit müssen die ersten k Koordinaten unterschiedlich sein. Aber α könnte man auch so definieren, dass irgendwelche anderen $d - 1 = n - k$ Stellen abgeschnitten werden. Dann müssen mit demselben Argument auch die übrig gebliebenen k Koordinaten unterschiedlich sein.

5.2.3 Beispiel: MDS-Codes

Ein kleiner MDS-Code mit Parameter $[4, 3, 2]_5$
Wir betrachten den linearen Code C über $\mathbb{Z}_5$ mit Generatormatrix

$$G = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 \\ 1 & 4 & 4 & 1 \end{pmatrix}.$$

Nach Abschn. 2.2 hat er Parameter $[4, 3, 2]_5$. Also gilt $d = 2 = 4 - 3 + 1 = n - k + 1$ und C ist ein MDS-Code.

MDS- und Hamming-Codes
Jetzt versuchen wir es mal mit Hamming-Codes $Ham_q(m)$ der Länge $n = (q^m-1)/(q-1)$, der Dimension $k = n - m$ und mit Minimalabstand $d = 3$. Somit ist $Ham_q(m)$ genau dann ein MDS-Code, wenn $3 = d = n - k + 1 = m + 1$ gilt, also wenn $m = 2$ und damit $n = q + 1$ ist.

Binäre MDS-Codes
Wir haben nicht umsonst als erstes Beispiel einen Code über einem *größeren* Körper gewählt. Während wir bislang im Wesentlichen mit binären und gelegentlich ternären Codes ausgekommen sind, erfordern die *guten* MDS-Codes größere endliche Körper. Es gilt nämlich folgende Aussage. Sei C ein binärer $[n, k, d]$-MDS-Code mit $0 \subset C \subset (\mathbb{Z}_2)^n$. Dann ist $C = \{(0, \ldots, 0), (1, \ldots, 1)\}$ ein Wiederholungscode mit $k = 1$ und $n = d$ oder es ist $C = \{(c_1, \ldots, c_n) \in (\mathbb{Z}_2)^n \,|\, c_1 + \ldots + c_n = 0\}$ ein Paritätsprüfungscode mit $k = n - 1$ und $d = 2$.

Es gibt also keine interessanten binären MDS-Codes. MDS-Codes sind aber – im Gegensatz zu den perfekten Codes – nicht vollständig klassifiziert.

5.2.4 Verallgemeinerte Reed-Solomon-Codes

Das folgende Konstruktionsprinzip liefert die heutzutage wohl besten und folglich die am weitesten verbreitetste Familie von Blockcodes.

Sei K ein Körper mit $|K| = q$ und $1 \leq k \leq n \leq q$. Sei außerdem $K[x]$ die Menge aller Polynome über K mit der Variablen x und $K[x]_{k-1}$ der k-dimensionale Vektorraum der Polynome vom Grad $\leq k - 1$. Weiter sei $\mathfrak{M} = \{\alpha_1, \alpha_2, \ldots, \alpha_n\} \subseteq K$ mit verschiedenen α_i. Wir nennen dann $RS(k, \mathfrak{M}) = \{(f(\alpha_1), f(\alpha_2) \ldots, f(\alpha_n)) | f(x) \in K[x]_{k-1}\}$ einen **verallgemeinerten Reed-Solomon-Code**.

Reed-Solomon-Codes wurden 1960 von **Irving S. Reed** (den wir schon von den Reed-Muller-Codes und dem Majority-Logic-Algorithmus her kennen) und dem amerikanischen Mathematiker **Gustave Solomon** (1930–1996) am MIT Lincoln Laboratory, einer Forschungseinrichtung des Verteidigungsministeriums der Vereinigten Staaten, entwickelt. Die Publikation von Reed und Solomon mit dem Titel „Polynomial codes over certain finite fields“ [ReSo] stellt grundsätzlich den nächsten Meilenstein in der Geschichte der Codierungstheorie dar, mit einem Schönheitsfehler: Zu dieser Zeit war die praktische Verwendbarkeit dieser Codes eingeschränkt, da noch keine effiziente Methode zur

Decodierung bekannt war. Ein effizienter Decodieralgorithmus war erst ca. zehn Jahre später verfügbar. Darauf kommen wir natürlich auch zu sprechen, aber erst in den Abschn. 7.4 und Abschn. 7.5. Jetzt aber ganz langsam der Reihe nach zu den Eigenschaften der verallgemeinerten Reed-Solomon-Codes.

$RS(k, \mathfrak{M})$ hat Länge n und ist bis auf Äquivalenz eindeutig.

Das ist jedenfalls mal klar, denn es handelt sich offenbar um n-Tupel und die Reihenfolge der α_i ist willkürlich gewählt.

$RS(k, \mathfrak{M})$ ist ein linearer Code über K, also ein Unterraum des Vektorraums der n-Tupel K^n über K.

Sei dazu $c = (f(\alpha_1), f(\alpha_2) \ldots., f(\alpha_n)), c' = (g(\alpha_1), g(\alpha_2) \ldots., g(\alpha_n)) \in RS(k, \mathfrak{M})$ und $a, a' \in K$. Wir müssen uns überlegen, dass dann auch $ac + a'c' \in RS(k, \mathfrak{M})$ ist. Dazu benutzen wir aber einfach das Polynom $h(x) = af(x) + a'g(x) \in K[x]_{k-1}$ und folgern $(h(\alpha_1), h(\alpha_1) \ldots., h(\alpha_n)) \in RS(k, \mathfrak{M})$.

$RS(k, \mathfrak{M})$ hat Dimension k.

Wir machen hierzu einen kleinen Trick. Wir überlegen uns nämlich, dass für zwei verschiedene $f(x), g(x) \in K[x]_{k-1}$ auch die zugehörigen Codewörter $c = (f(\alpha_1), \ldots, f(\alpha_n))$ und $c' = (g(\alpha_1), \ldots, g(\alpha_n))$ unterschiedlich sind. Wäre dies nämlich nicht der Fall, so wäre $c - c' = ((f-g)(\alpha_1), \ldots, (f-g)(\alpha_n)) = 0$ und das Polynom $f(x) - g(x) \neq 0$ hätte n verschiedene Nullstellen $\alpha_1, \ldots, \alpha_n$. Das kann aber nicht sein, da $\text{grad}(f(x) - g(x)) \leq k - 1 < n$ ist. Damit gilt $q^k = |K[x]_{k-1}| \leq |RS(k, \mathfrak{M})|$. Umgekehrt kann nach Definition $RS(k, \mathfrak{M})$ nur höchstens $|K[x]_{k-1}|$ Elemente haben. Also gilt die Gleichheit und $\dim(RS(k, \mathfrak{M})) = k$.

$RS(k, \mathfrak{M})$ hat Minimalabstand $d = n - k + 1$.

Wir versuchen es gleich noch mal mit dem Nullstellenargument von eben. Jedes Codewort $0 \neq c = (f(\alpha_1), \ldots, f(\alpha_n)) \in RS(k, \mathfrak{M})$ hat nämlich den Wert 0 an höchstens $k - 1$ seiner Positionen, da $f(x)$ wegen $\text{grad}(f(x)) \leq k - 1$ höchstens $k - 1$ Nullstellen hat. Dann hat also c an mindestens $n - (k - 1) = n - k + 1$ Positionen einen Wert $\neq 0$ und wir haben uns damit $d \geq n - k + 1$ überlegt. Andererseits gilt für das Polynom $g(x) = (x - \alpha_1)(x - \alpha_2) \ldots .(x - \alpha_{k-1}) \in K[x]_{k-1}$, dass $g(x)$ genau die Nullstellen $\alpha_1, \alpha_2, \ldots, \alpha_{k-1}$ hat. Somit hat das Codewort $c' = (g(\alpha_1), g(\alpha_2) \ldots., g(\alpha_n)) \in RS(k, \mathfrak{M}))$ an genau $n - (k - 1) = n - k + 1$ Positionen einen Wert $\neq 0$. Also ist $wt(c') = n - k + 1$ und somit $d = n - k + 1$.

$RS(k, \mathfrak{M})$ ist ein MDS-Code.

Genau das haben wir in eben hergeleitet, die Singleton-Schranke wird angenommen.

Die Matrix

$$G = \begin{pmatrix} 1 & 1 & 1 & \ldots & 1 \\ \alpha_1 & \alpha_2 & \alpha_3 & \ldots & \alpha_n \\ \alpha_1^2 & \alpha_2^2 & \alpha_3^2 & \ldots & \alpha_n^2 \\ \vdots & \vdots & \vdots & & \vdots \\ \alpha_1^{k-1} & \alpha_2^{k-1} & \alpha_3^{k-1} & \ldots & \alpha_n^{k-1} \end{pmatrix}$$

ist eine Generatormatrix von $RS(k, \mathfrak{M})$.

Jedes Codewort c aus dem von G erzeugten Code lässt sich schreiben als Linearkombination der Zeilenvektoren $z_0, z_1, \ldots, z_{k-1}$ von G, sagen wir, $c = a_0 z_0 + \ldots + a_{k-1} z_{k-1}$ mit geeigneten $a_i \in K$. Wir betrachten das Polynom $f(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_{k-1} x^{k-1} \in K[x]_{k-1}$. Dann lässt sich c aber auch schreiben als $c = (f(\alpha_1), f(\alpha_2), f(\alpha_3), \ldots, f(\alpha_n))$ und ist folglich auch ein Codewort in $RS(k, \mathfrak{M})$.

Die Kontrollmatrix eines Reed-Solomon-Codes benötigen wir zurzeit nicht. Wir kommen aber in Abschn. 7.1 darauf zurück.

5.2.5 Reed-Solomon-Codes – die wahren Stars

Wie nicht anders zu erwarten, stellen sich *normale* Reed-Solomon-Codes als spezielle verallgemeinerte Reed-Solomon-Codes heraus. Jedenfalls sind diese die wahren Stars unter den Blockcodes.

Bei verallgemeinerten Reed-Solomon-Codes wählt man speziell $n = q - 1$, $\mathfrak{M} = \{\alpha_1, \alpha_2, \ldots, \alpha_{q-1}\} = K^* = K \backslash \{0\}$ und anstelle des Parameters k mit $1 \leq k \leq n$ nimmt man $d = n - k + 1 = q - k$. Dann heißt

$$RS_q(d) = RS(q - d, K^*) = \{(f(\alpha_1), f(\alpha_2) \ldots, f(\alpha_{q-1})) | f(x) K[x]_{q-d-1}\}$$

Reed-Solomon-Code.
$RS_q(d)$ ist ein MDS-Code und hat die Parameter $[q - 1, q - d, d]_q$.

5.2.6 Beispiel: Reed-Solomon-Codes

Bevor wir nun wieder ausführlich zu Beispielen kommen, sei nochmals explizit darauf hingewiesen, dass wegen $|K| = q = n + 1$ Reed-Solomon-Codes nur über größeren Körpern Sinn machen.

Reed-Solomon-Code $RS_5(2)$
Für $RS_5(2)$ gilt $q = 5, n = q - 1 = 4, k = n - d + 1 = 3$ und $(\mathbb{Z}_5)^* = \{1, 2, 3, 4\}$. Hier ist die Generatormatrix:

$$G = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 \\ 1 & 2^2 = 4 & 3^2 = 4 & 4^2 = 1 \end{pmatrix}.$$

Wir stellen fest, dass unser eingangs beschriebener MDS-Code über $\mathbb{Z}_5$ genau $RS_5(2)$ ist.

Reed-Solomon-Code $RS_7(3)$
Für $RS_7(3)$ gilt $q = 7, n = q - 1 = 6, k = n - d + 1 = 4$ und $(\mathbb{Z}_7)^* = \{1, 2, 3, 4, 5, 6\}$. Hier ist wieder die Generatormatrix:

$$G = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 4 & 2 & 2 & 4 & 1 \\ 1 & 1 & 6 & 1 & 6 & 6 \end{pmatrix}.$$

Reed-Solomon-Code $RS_4(2)$
Für $RS_4(2)$ gilt $q = 2^2, n = q - 1 = 3, k = n - d + 1 = 2$. Wir wissen aus dem letzten Abschnitt, dass wir uns den Körper K mit vier Elementen vorstellen können als $K = \{0, 1, t, t + 1\}$ mit zugehöriger Additions- und Multiplikationstafel. Also ist $K^* = \{1, t, t + 1\}$ und hier ist die Generatormatrix:

$$G = \begin{pmatrix} 1 & 1 & 1 \\ 1 & t & t+1 \end{pmatrix}.$$

Reed-Solomon-Code $RS_8(4)$
Für $RS_8(4)$ gilt $q = 2^3, n = q - 1 = 7, k = n - d + 1 = 4$ und $K^* = \{1, t, t + 1, t^2, t^2 + 1, t^2 + t, t^2 + t + 1\}$. Hier ist wieder die Generatormatrix:

$$G = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & t & t+1 & t^2 & t^2+1 & t^2+t & t^2+t+1 \\ 1 & t^2 & t^2+1 & t^2+t & t^2+t+1 & t & t+1 \\ 1 & t+1 & t^2 & t^2+1 & t^2+t & t^2+t+1 & t \end{pmatrix}.$$

5.2.7 Güte von Reed-Solomon-Codes

Schon die verbreitete Anwendung der Reed-Solomon-Codes, die wir im nächsten Abschnitt vorstellen werden, zeigt, dass es sich dabei um das Beste handelt, was als Blockcode verfügbar ist. Wegen der MDS-Eigenschaft $n = k + d - 1$ kann man sich stets zu vorgegebener Länge n die gewünschte Relation zwischen Rate (also k) einerseits und Korrekturkapazität (also d) andererseits auswählen. Der Nachteil von Reed-Solomon-Codes ist allerdings der, dass man stets mit größeren Körpern arbeiten muss. Wie man das umgehen kann, sehen wir in Abschn. 7.2 bei den sog. BCH-Codes.

Hier noch kurz zur asymptotischen Betrachtung von $RS_q(d)$. Bei vorgegebenem konstanten Minimalabstand d geht die Rate $k/n = (q-d)/(q-1)$ mit wachsender Codelänge und damit für größere Körper schnell gegen 1. Für die Fehlerkorrekturkapazität $e = 10$ beispielsweise wählt man $d = 21$. Für den Körper K mit $q = 2^8$ Elementen ist die Rate $(q - d)/(q - 1) = 235/255$ bereits über 90 %.

5.2.8 Auswertungscodes

Wir wollen zum Ende dieses Abschnitts noch ein Wort über das den Reed-Solomon-Codes zugrunde liegende Konstruktionsprinzip verlieren. Reed-Solomon-Codes gehören zur Klasse der sog. **Auswertungscodes**, d. h. die Codewörter sind Auswertungen von Polynomen an bestimmten Stellen. Der Vorteil hierbei ist, dass man Eigenschaften der Polynome direkt für die Bestimmung der Codeparameter nutzen kann, wie wir oben gesehen haben. Auch Reed-Muller-Codes sind Auswertungscodes und werden in der Literatur auch häufig als solche definiert. Allerdings handelt es sich dabei nicht um Polynome in $\mathbb{Z}_2[x]$ mit nur einer Variablen x, sondern um solche mit mehreren Variablen in $\mathbb{Z}_2[x_1, \ldots, x_m]$. Wie bei Reed-Solomon-Codes auch verwendet man dabei nur eine Teilmenge dieser Polynome, nämlich genau diejenigen, bei denen jede Variable x_i höchstens in erster Potenz auftritt und die Gesamtgrad $\leq r$ besitzen. Setzt man in ein solches Polynom alle binären m-Tupel für die Variablen $x_1, \ldots, x_m$ ein (d. h. man wertet die Polynome dort aus), so entspricht diese Beschreibung der Reed-Muller-Codes $\mathrm{RM}(r, m)$ weitgehend der mittels Boolescher Algebra. Mit unserer *einfacheren* Definition der Reed-Muller-Codes über die Plotkin-Konstruktion wurde diese weitere algebraische Abstraktionsebene vermieden.

Wie schon oben erwähnt, wurden effiziente Decodiermethoden für Reed-Solomon-Codes erst ca. zehn Jahre nach deren Publikation vervollständigt. Also müssen auch wir uns darauf noch etwas gedulden. Wir wollen uns vielmehr zunächst um Anwendungsfälle von Reed-Solomon-Codes kümmern.

5.3 Einige Anwendungen der Reed-Solomon-Codes

Die Entdeckung von Reed-Solomon-Codes war so etwas wie eine Revolution in der Codierungstheorie. Es gibt heute wenige Kommunikationskanäle oder Datenspeicher, bei denen kein Reed-Solomon-Code oder ein Derivat davon zumindest beteiligt ist. Reed-Solomon-Codes werden häufig auch in Kombination mit **Faltungscodes** eingesetzt, in sog. *Hybrid*verfahren, auf die wir aber erst in Kap. 9 eingehen können. Hier soll ein erster Eindruck der vielen Anwendungsfälle gegeben werden.

5.3.1 Anwendung: Voyager 2 – Uranus und Neptun

Wie bereits in Abschn. 3.4 besprochen, wurde die Codierungssoftware der Sonde **Voyager 2** auf ihrem Weg von Saturn zu Uranus und Neptun umprogrammiert von Golay G_{24} auf einen Reed-Solomon-Code, genauer $RS_{256}(33)$ von Länge $n = 255$ und Dimension $k = 255-33+1 = 223$. $RS_{256}(33)$ hat also über dem Körper K mit $2^8 = 256$ Elementen genau $(2^8)^{223} = 2^{1784}$ Codewörter. Ähnlich der alten Golay-Codierung wurden dabei im aus Farbwerten bestehenden Informationsstring jeweils 1784 Bits zu einem Informations-

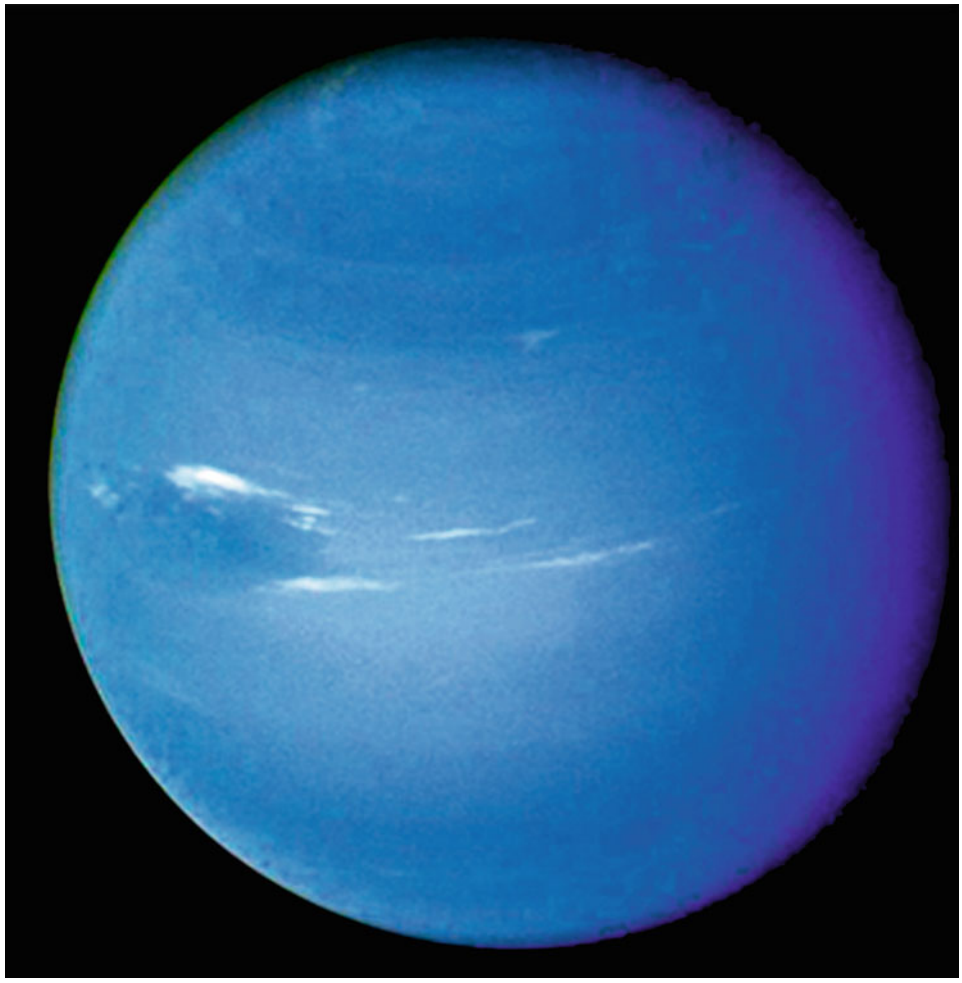


Abb. 5.1 Der Planet Neptun, aufgenommen von Voyager 2 (Quelle NASA [WPVo2])

block zusammengefasst, je 8 Bits als Körperelemente in K interpretiert und anschließend gemäß $RS_{256}(33)$ codiert. Die Rate $k/n = 223/255$ ist ungefähr $6/7$ und damit bei Weitem besser als $1/2$ beim Golay-Code G_{24}. Wegen $d = 33$ kann $RS_{256}(33)$ bis zu 16 Fehler korrigieren gegenüber 5 bei G_{24}. Während man allerdings bei Golay G_{24} in einfacher Weise mit binären Ziffern rechnen konnte, muss man beim Körper K mit seinen 2^8 Elementen dessen Additions- und Multiplikationstafel heranziehen, die wir im vorletzten Abschnitt besprochen haben.

Dies war allerdings noch nicht die volle Wahrheit. Die Umstellung erfolgte nicht von G_{24} auf $RS_{256}(33)$ allein, sondern auf einen Code, der aus einer Verkettung von $RS_{256}(33)$ mit einem Faltungscode besteht. Jedoch wurde letzterer auch schon in Kombination mit

G_{24} genutzt. Dazu müssen wir uns zuerst in Kap. 9 mit Faltungscodes beschäftigen. Zuvor aber vermittelt uns Abb. 5.1 schon mal einen Eindruck der Fotos von Voyager 2.

5.3.2 Anwendung: Galileo und Cassini – Jupiter und Saturn

Galileo war eine amerikanische Raumsonde der NASA, die den Planeten Jupiter und seine Monde näher erkunden sollte. Sie wurde 1989 verspätet mit dem Spaceshuttle Atlantis erfolgreich gestartet; die *Challenger-Katastrophe* drei Jahre zuvor hatte diese Verzögerung verursacht. Galileo erreichte Jupiter erst 1995 nach einem Umweg wegen ungünstiger Planetenkonstellation vorbei an Venus, um sich dort gravitativen Schwung für die Reise zum Jupiter zu holen. Es war der erste Orbiter in einer Jupiterumlaufbahn. Nach vielen Jahren erfolgreicher Arbeit wurde die Mission 2003 beendet und die Sonde geplant in den Jupiter stürzen gelassen.

Cassini-Huygens ist der Name einer NASA-Mission zur Erforschung des Planeten Saturn und seiner Monde. Bei Cassini handelt es sich um eine Orbiter, um die Himmelskörper aus einer Umlaufbahn um den Saturn zu untersuchen. Huygens wurde als Landungssonde konzipiert, um von Cassini abgekoppelt auf dem Mond Titan zu landen und diesen mittels direkter Messungen in der Atmosphäre und auf der Oberfläche zu erforschen, was wegen der dichten und schwer zu durchdringenden Atmosphäre des Mondes nicht von einer Umlaufbahn aus möglich ist. Die beiden aneinandergekoppelten Sonden wurden 1997 gestartet und 2004 schwenkte Cassini in die Umlaufbahn um den Saturn ein. Im Jahr 2005 landete Huygens nach der Trennung von Cassini auf Titan und sandte 72 min lang Daten, die das Verständnis über den Mond erheblich verbesserten.

Bemerkenswert ist hierbei, dass beide Missionen bei der Datenübertragung zur Erde weiterhin im Wesentlichen die gleiche Konstellation an fehlerkorrigierenden Codes verwandt haben wie Voyager 2, nämlich den Reed-Solomon-Code $RS_{256}(33)$ verkettet mit einem – allerdings jeweils deutlich veränderten – Faltungscode. Abb. 5.2 vermittelt einen künstlerischen Gesamteindruck der Cassini-Huygens-Mission.

5.3.3 Anwendung: Mars-Rover

Auch bei den neueren Mars-Missionen, bei denen jeweils **Mars-Rover** gelandet wurden, basiert das Fehlerkorrekturverfahren bei der Datenübertragung weiterhin auf Reed-Solomon-Codes. Weitere Details hierzu folgen in Abschn. 9.3.

5.3.4 Anwendung: 2-D-Barcodes

Ein **2-D-Barcode** besteht aus einer in der Regel quadratischen Matrix aus kleinen schwarzen und weißen Quadraten. Hierbei wird also der Begriff **Code** im Sinne einer Quellenco-

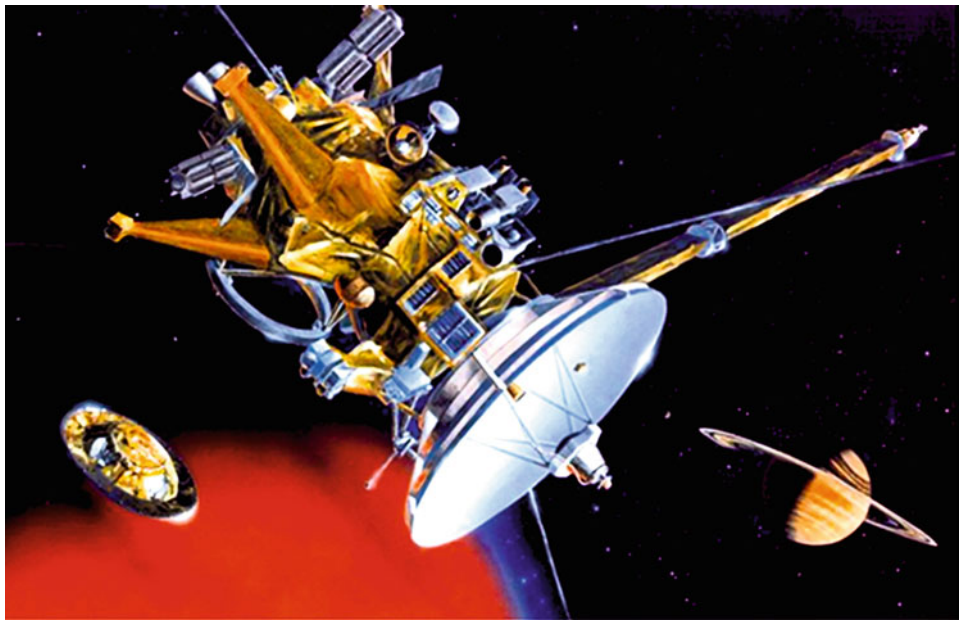


Abb. 5.2 Künstlerische Darstellung von Cassini (große Sonde) und Huygens (links) vor dem Mond Titan (Vordergrund) und Saturn mit seinen Ringen (rechts) (Quelle NASA [WPCaH])

dierung verwendet. Es handelt sich technisch gesehen lediglich um eine Fortentwicklung des normalen 1-D-Barcodes, der sich etwa als EAN auf Produkten befindet, allerdings mit wesentlich mehr Möglichkeiten und Funktionalitäten. Auf dem Markt befinden sich verschiedene Entwicklungen, z. B. **QR (Quick Response)**, **DataMatrix** und **Aztec**. Smartphones verfügen über eine eingebaute Kamera und entsprechende Software – meist Apps zum Downloaden –, die das Interpretieren von 2-D-Barcodes ermöglicht. 2-D-Barcodes sind vielfach im Einsatz, z. B. in der Produktionslogistik (z. B. QR bei Toyota), im Marketing, als Fahrplanauskunft und Navigationshilfe und als mobile Visitenkarte. Die Deutsche Post nutzt DataMatrix zum Freimachen von Briefen, die Deutsche Bahn sowie einige Fluglinien Aztec für deren Onlinetickets. Die Daten aller drei genannten 2-D-Barcodes sind mittlerweile durch fehlerkorrigierende Reed-Solomon-Codes geschützt.

Aztec-Code

Wir betrachten hier zur Illustration den Aztec-Code etwas genauer. Abb. 5.3 zeigt ein Beispiel.

Aztec wurde 1995 in den USA entwickelt und ist heute frei verfügbar. Es können dabei kleine bis große Datenmengen (etwa 2000 Bytes) codiert werden. In der Regel werden dabei ASCII-Zeichen benutzt. Die Reed-Solomon-Fehlerkorrektur erlaubt die Rekonstruktion des Dateninhaltes auch dann noch, wenn bis zu 25 % (bei kleinen Codes

Abb. 5.3 Beispiel eines Aztec-Codes (Quelle [WPAzt])

sogar bis zu 40 %) des Codes zerstört worden sind. Der Anteil des Barcodes, welcher nicht für Informationsinhalte verwendet wird, kann für Reed-Solomon-basierte redundante Daten genutzt werde. Die Aufteilung ist anwendungsspezifisch frei konfigurierbar, empfohlen wird im Mittel ein Anteil von 25 % Redundanz. Da Reed-Solomon-Codes MDS-Codes sind (d. h. $n = k + d - 1$ gilt), entspricht dies einer Fehlerkorrekturrate von ca. 12 %. Der Name Aztec-Code ist der Tatsache geschuldet, dass er aussieht wie eine Aztekenpyramide von oben. Im Mittelpunkt des Codes befindet sich das Suchelement, das aus mehreren ineinander verschachtelten Quadraten besteht. Die eigentliche digitalisierte Information wird dann spiralförmig um das *Bullauge* im Zentrum aufgetragen, und zwar in sog. **Layer**. Jeder weitere Layer umrundet den vorhergehenden und auf diese Weise wächst die Datenmenge des Barcodes ständig an.

Wie wird nun mittels Reed-Solomon codiert? Jeder Aztec-Code enthält den sog. **Modus**. Dieser setzt sich zusammen aus der binär dargestellten Anzahl der Layer und der Informationszeichen im Aztec-Quadrat; das sind insgesamt 16 Modusbits. Diese werden als vier Elemente im Körper K mit $2^4 = 16$ Elementen aufgefasst und mittels eines verallgemeinerten Reed-Solomon-Codes um sechs Stellen auf zehn Stellen über dem Körper K erweitert, also auf insgesamt 40 Bits. Der Modus markiert den Anfang in der oben genannten Datenspirale. Wegen $10 = n = k + d - 1 = d + 3$ können drei Fehler im Modus korrigiert werden. Bei den größeren Aztec-Codes werden dann jeweils 12 Bits an Information als Elemente im Körper K mit $2^{12} = 4096$ Elementen aufgefasst und anwendungsspezifisch k solcher Zeichen in einen verallgemeinerten Reed-Solomon-Code der Länge $n \leq 2^{12}-1 = 4095$ gepackt. Dabei wird $d = n-k+1$ entsprechend der gewünschten Korrektureigenschaft gewählt. Anstelle mit verallgemeinerten Reed-Solomon-Codes zu arbeiten, kann man auch normale Reed-Solomon-Codes verkürzen, wie wir im nächsten Abschnitt sehen werden. Jedenfalls geht die Gesamtkonstruktion noch einher mit diversen Füllbits (sog. *Bit-Stuffing* und *-Padding*), auf die wir nicht genauer eingehen wollen. Tab. 5.1 enthält zusammengefasst die für den Modus und die jeweilige Layer-Anzahl verwendeten Körper sowie die irreduziblen Polynome, über die die Körpererweiterungen definiert sind.

DataMatrix und QR-Code

DataMatrix wurde erst in neueren Versionen von Faltungscodes auf Reed-Solomon-Codes umgestellt. Wir wollen auf den DataMatrix-Code aber nicht weiter eingehen.

Tab. 5.1 Parameter für die Reed-Solomon-Codes des Aztec-2-D-Barcodes (Quelle [WPAzt])

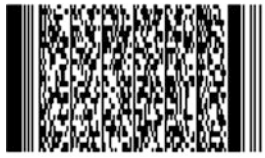
Bits	Körper	Polynom	Benutzt bei
4	$2^4 = 16$	$x^4 + x + 1$	Modus
6	$2^6 = 64$	$x^6 + x + 1$	1–2 Layer
8	$2^8 = 256$	$x^8 + x^5 + x^3 + x^2 + 1$	3–8 Layer
10	$2^{10} = 1024$	$x^{10} + x^3 + 1$	9–22 Layer
12	$2^{12} = 4096$	$x^{12} + x^6 + x^5 + x^3 + 1$	23–32 Layer

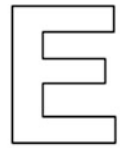
Sobald wir jedoch noch mehr über Reed-Solomon- und auch BCH-Codes erfahren haben, werden wir in Abschn. 7.3 die Codierung von **QR-Codes** im Detail besprechen.

5.3.5 Anwendung: Der gestapelte Barcode PDF 417

Der Barcode **Portable Data File PDF 417** ist – entgegen der weitverbreiteten Meinung – kein richtiger zweidimensionaler 2-D-Barcode. Er ist vielmehr aus mehreren aufeinander gestapelten Zeilen (maximal 90) aufgebaut, die jede für sich wie lineare Barcodes aussehen, angeordnet auf einem rechteckigen Feld. Deshalb nennt man PDF 417 auch einen **gestapelten Barcode**. Eine der bekanntesten Anwendungen von PDF 417 sind die Bordkarten der Lufthansa, wie das Beispiel in Abb. 5.4 zeigt.

Jede Zeile enthält links und rechts diverse Rand- und Steuerzeichen sowie jeweils dazwischen maximal 30 eigentliche Datenzeichen – die sog. **Symbol Characters**. Jedes Datenzeichen wird dabei aus vier schwarzen Strichen und vier Lücken quellencodiert, die insgesamt eine Breite von 17 sog. **Modulen** haben. Daher stammt auch die Bezeichnung 417. Abb. 5.5 zeigt ein entsprechendes Beispiel.





Buchungscode:**Y7AJLJ**

API
Boarding Pass

Name	**MANZ / OLAF DR MR**		
Flug	**LH0760 / 06.Nov 15** Frankfurt - Delhi		
Abfluggate	C16	Terminal 1	
Boardingzeit	**12:45**	Boarding Nummer	**0173**
Abflugzeit	13:30	Fluggesellschaft	LUFTHANSA
Sitznummer	51K	etix	220 9716543244
Klasse	Premium Economy	Passagier Status	FTL
Gepäckabgabe	**Gepäckautomat**	Gepäck	

Abb. 5.4 Bordkarte der Lufthansa mit dem Barcode PDF 417

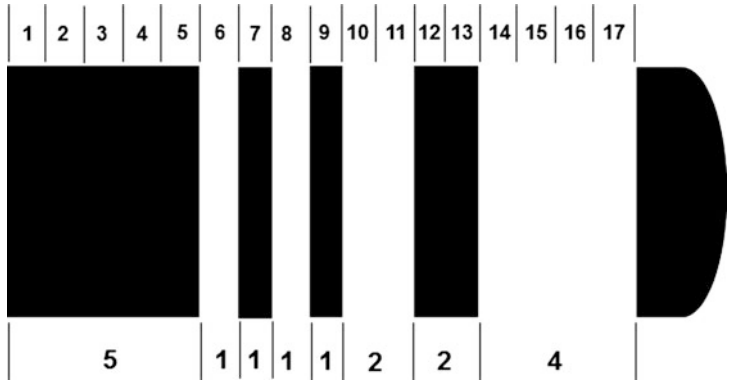


Abb. 5.5 Beispiel eines Datenzeichens bei PDF 417

Auch bei PDF 417 wird die Fehlerbehandlung mit Reed-Solomon-Codes realisiert, und zwar in neun wählbaren Fehlerkorrekturstufen (Error Correction Levels) von 0 bis 8. Wir wollen auch dieses Verfahren genauer erläutern. Hierzu benötigen wir aber etwas mehr algebraischen Hintergrund und kommen deshalb erst in Abschn. 7.1 wieder darauf zurück.

5.3.6 Anwendung: MIDS – NATO-Militärfunk

MIDS, Multifunctional Information Distribution System (dt. multifunktionales Informationsverteilungssystem), ist ein militärisches Funksystem im Frequenzbereich 890–1215 MHz. Ursprünglich für die Streitkräfte der USA entwickelt, wird das System heute u. a. im Gesamtbereich der NATO-Luftverteidigung genutzt. MIDS ist die verbesserte Version des von den USA entwickelten JTIDS und mit diesem weitgehend kompatibel. Die wesentlichen Leistungsmerkmale sind:

- Datenfunk,
- Sprachübertragung,
- Flugnavigation,
- Personen-/Ortsidentifizierung.

Zur Geheimhaltung nutzt MIDS natürlich in erster Linie diverse Chiffriermethoden. Aber es kommt auch zusätzlich ein Fehlerkorrekturverfahren mittels Reed-Solomon-Codes zum Einsatz, nämlich $RS_{32}(17)$ von Länge $n = 31$ über dem Körper K mit $2^5 = 32$ Elementen. Wegen $k = n - d + 1 = 31 - 17 + 1 = 15$ ergibt dies eine Rate $k/n = 15/31$ von ca. $1/2$. Wegen $d = 17$ kann der Code bis zu acht Fehler korrigieren.

5.3.7 Anwendung: Digitalfernsehen mit DVB

Reed-Solomon-Codes werden auch bei der Ausstrahlung des Digitalfernsehens eingesetzt. Der zugehörige Standard heißt **DVB (Digital Video Broadcasting).** Genauer gesagt wird auch hier wieder ein Reed-Solomon-Code mit einem Faltungscode verkettet, sodass wir darauf erst in Abschn. 8.3 und 9.6 – aber dann im Detail – zurückkommen wollen.

5.3.8 Anwendung: Telefonie und schnelles Internet mit DSL

DSL (Digital Subscriber Line) ist ein Übertragungsstandard, bei dem Daten mit hohen Übertragungsraten über Telefonleitungen gesendet und empfangen werden können. Auch hierbei kommen zur automatischen Fehlerkorrektur Reed-Solomon-Codes zum Einsatz. Weitere Einzelheiten vertagen wir ebenfalls auf Abschn. 9.6.

Stattdessen wollen wir uns nun um eine weitere sehr wichtige Anwendung der Reed-Solomon-Codes kümmern, nämlich die Speicherung von Daten auf **Audio-CDs** und **Daten-DVDs**.

5.4 Verkürzte Codes und Cross-Interleaving

Wir kommen aber zunächst auf zwei weitere Konstruktionsprinzipien für *gute* Codes zu sprechen, die Verkürzung und das Interleaving.

5.4.1 Der verkürzte Code

Sei C ein linearer $[n,k,d]_q$-Code über dem Körper K und $n > 1$. Dann heißt der Code $C^\circ = \{(a_1,\ldots,a_{n-1}) | (a_1,\ldots,a_{n-1},0) \in C\} \subseteq K^{n-1}$ der **verkürzte Code** zu C.

Das Verkürzen kann man natürlich auch wiederholt ausführen bzw. auch jede andere Position entsprechend weglassen.

C° ist natürlich wieder ein linearer Code und hat Länge $n-1$.

Wir überlegen uns nun, dass $\dim(C^\circ) = k$ oder $k-1$ ist. Sei dazu $c_1,\ldots,c_k$ eine Basis von $C \subseteq K^n$. Wir schreiben $c_i = (v_i, p_i)$, wobei die v_i die ersten $n-1$ Positionen von c_i umfassen und die p_i die letzte. Sind alle $p_i = 0$, so sind die v_i eine Basis von C° und $\dim(C^\circ) = k$. Sei also etwa $p_1 \neq 0$. Wir können dann Vielfache von c_1 derart zu den übrigen c_i addieren, dass deren letzte Komponente gleich 0 wird. Diese neuen Vektoren bilden immer noch eine Basis von C. Wir nennen sie der Einfachheit halber wieder c_i, wissen aber jetzt, dass alle c_i außer c_1 an der letzten Position eine 0 haben. Da

die $c_2, \ldots, c_k$ linear unabhängig sind, gilt dies auch für $v_2, \ldots, v_k$ und diese liegen auch in C°. Also ist in diesem Fall $\dim(C^\circ) \geq k-1$.

Letztlich ist $d(C^\circ) \geq d(C)$ für $k > 1$. Sei dazu wieder $0 \neq c = (v, p) \in C$, wobei v die ersten $n-1$ Positionen umfasst und p die letzte. Ist $p = 0$, so ist $0 \neq v \in C^\circ$ und $wt(v) = wt(c)$. Ist dagegen $p \neq 0$, so ist $v \notin C^\circ$ und hat daher keinen Einfluss bei der Berechnung von $d(C^\circ) = \min\{wt(u) | u \in C^\circ\}$. Also folgt $d(C^\circ) \geq d(C)$.

5.4.2 Beispiel: Verkürzte Codes

Hier ist also der typische Einsatz von Codeverkürzungen. Man wählt zunächst einen *guten* $[n, k]$-Code C, z. B. einen Reed-Solomon-Code $RS_q(d)$. Für eine konkrete Anwendung will man aber nur k'-Tupel systematisch codieren mit $k' < k$. Dann lässt man bei C einfach die ersten $k - k'$ Positionen weg, die ja für die Anwendung ohnehin 0 sind, d. h. man verkürzt den Code einfach oder mehrfach und erhält einen passenden kürzeren Code mit mindestens gleicher Fehlerkorrekturkapazität.

MDS-Code C_1 mit Parameter $[32, 28, 5]_{256}$,
MDS-Code C_2 mit Parameter $[28, 24, 5]_{256}$
Wir starten mit dem Reed-Solomon-Code $C = RS_{256}(5)$ und dessen Parameter $[255, 251, 5]_{256}$. Jetzt kürzen wir diesen und erhalten einen $[254, k, d]$-Code C° mit $k = 251$ oder 250 und $d \geq 5$. Aus der Singleton-Schranke ergibt sich andererseits $5 \leq d \leq 254 - k + 1$, folglich $k = 250$ und $d = 5$. Also ist C° ein MDS-Code mit Parameter $[254, 250, 5]$. Dieses Verfahren können wir jetzt mehrfach wiederholen, bis wir die Codes C_1 und C_2 bekommen. Warum gerade diese? Die Auflösung findet sich im nächsten Abschnitt bei Audio-CDs.

MDS-Code C_3 mit Parameter $[182, 172, 11]_{256}$,
MDS-Code C_4 mit Parameter $[208, 192, 17]_{256}$
Wir wenden das gleiche Verfahren diesmal auf die Reed-Solomon-Codes $RS_{256}(11)$ und $RS_{256}(17)$ an. Diese haben Parameter $[255, 245, 11]_{256}$ bzw. $[255, 239, 17]_{256}$. Wir kürzen die beiden Codes und erhalten mittels Singleton-Schranke zwei MDS-Codes mit Parameter $[254, 244, 11]$ bzw. $[254, 238, 17]$. Wiederholtes Kürzen führt schließlich auf die Codes C_3 und C_4. Diese spielen bei Daten-DVDs eine Rolle, wie ebenfalls der nächste Abschnitt zeigt.

MDS-Code C_5 mit Parameter $[204, 188, 17]_{256}$
Schließlich liefert das Kürzungsverfahren ebenfalls angewandt auf den Reed-Solomon-Code $RS_{256}(17)$ den Code C_5. Dieser wird bei DVB (Digital Video Broadcasting) eingesetzt.

Alle diese Codes hätten wir statt mit Codeverkürzung auch direkt konstruieren können, indem wir die entsprechenden verallgemeinerten Reed-Solomon-Codes $RS(k, \mathfrak{M})$ für ein

$\mathfrak{M} \subseteq K$ mit $|\mathfrak{M}| = n$ und $k = n - d + 1$ genommen hätten. Mittels Codeverkürzung kommt man aber in den Anwendungsfällen mit den normalen Reed-Solomon-Codes aus.

Erweiterter verkürzter Hamming-Code $Ham_2(7)$ mit Parameter [72, 64, 4]
Wir betrachten jetzt den binären Hamming-Code $Ham_2(7)$ mit Parameter $[127, 120, 3]_2$ und wollen diesen verkürzen, also die Parameter $[126, k, d]$ von $Ham_2(7)^\circ$ bestimmen. Jedenfalls gilt hierbei $119 \leq k \leq 120$ und $d \geq 3$. Jetzt benutzen wir die Kugelpackungsschranke für binäre Codes mit $q = 2$ und erhalten

$$2^{n-k} \geq \sum_{i=0}^{e} \binom{n}{i}.$$

Für die linke Seite ergibt sich in unserem Fall entweder $2^7 = 128$ oder $2^6 = 64$, für die rechte Seite $1 + 126 + \Delta$ mit $\Delta = 0$ oder $\Delta \geq 126 \cdot 125/2$. Dies zeigt $k = 119$ und $3 \leq d \leq 4$. Also ist $Ham_2(7)^\circ$ ein binärer $[126, 119, d]$-Code mit $d = 3$ oder 4. Diese Argumentation wiederholen wir wieder, bis wir einen binären $[71, 64, d]$-Code C erhalten mit $d = 3$ oder 4. Wir bilden dazu den erweiterten Code $C^\wedge$, also einen $[72, 64, d^\wedge]$-Code mit $d \leq d^\wedge \leq d + 1$. Wir wissen aber auch, dass ein erweiterter binärer Code gerades Gewicht hat, also ist $d^\wedge = 4$ und $C^\wedge$ ist ein $[72, 64, 4]$-Code. Wir haben damit – wie in Abschn. 3.2 versprochen – genau den Code konstruiert, der für ECC-Memories bei 64-Bit-Architektur eingesetzt wird.

Erweiterter verkürzter Hamming-Code $Ham_2(6)$ mit Parameter [39, 32, 4]
Nun betrachten wir den binären Hamming-Code $Ham_2(6)$ mit Parameter $[63, 57, 3]_2$ und wenden hierauf das gleiche Verfahren an. Dies führt durch sukzessive Verkürzung zunächst auf einen binären $[38, 32, d]$-Code mit $d = 3$ oder 4 und anschließend durch Erweiterung auf einen binären $[39, 32, 4]$-Code. Dies ist der Code, der für ECC-Memories bei 32-Bit-Architektur eingesetzt wird.

Erweiterter verkürzter Hamming-Code $Ham_2(5)$ mit Parameter [30, 24, 4]
Als Letztes noch der binäre Hamming-Code $Ham_2(5)$ mit Parameter $[31, 26, 3]_2$. Das obige Verfahren liefert durch Verkürzung zunächst einen binären $[29, 24, d]$-Code mit $d = 3$ oder 4 und durch Erweiterung einen $[30, 24, 4]$-Code. Diesen Code haben wird in Abschn. 3.2 im Zusammenhang mit dem GPS-Signal L1 C/A erwähnt.

5.4.3 Block- und Cross-Interleaving

Nachdem wir uns jetzt für diverse Anwendungen Codes der gewünschten Länge zurechtlegen können, müssen wir noch ein weiteres Problem *umschiffen*, das der **Fehlerbündelung**. Treten die Fehler nämlich vielfach gehäuft auf, so nutzt der beste lineare Code nichts. Denn dann werden entweder ein ganzes Codewort oder sogar mehrere hintereinander

fast komplett unbrauchbar, sodass wir nicht korrigieren können. Andererseits treten in den restlichen Codewörtern kaum weitere Fehler auf, sodass eine Codierung hier von vornherein unnötig ist. Wie kommen wir aus diesem Dilemma heraus? Die Lösung heißt Interleaving.

Block-Interleaving (**Spreizung**) ist eine Technik, die man zur Korrektur von Fehlerbündeln anwendet und für die man jeden linearen Code verwenden kann. Der Trick ist eigentlich ganz einfach.

Wir stellen uns dazu vor, dass wir eine Information mittels eines linearen Codes der Länge n senden oder auf einen Datenträger abspeichern wollen. Wir senden bzw. speichern dann die entsprechenden Codewörter $c_i = (c_{i1}, \ldots, c_{in})$ nicht wie gewohnt der Reihe nach, sondern schreiben zunächst – sagen wir – m Stück zeilenweise in eine temporäre Matrix.

$$\begin{pmatrix} c_{11} & \ldots & c_{1n} \\ \vdots & & \vdots \\ c_{m1} & \ldots & c_{mn} \end{pmatrix}$$

Den Matrixinhalt lesen wir dann spaltenweise aus, also n Wörter der Gestalt $c'_j = (c_{1j}, \ldots, c_{mj})$ jeweils von Länge m, und senden diesen String über den Kanal bzw. speichern ihn auf unseren Datenträger. Man spricht dabei von **Block-Interleaving** der **Tiefe** m. Der Empfänger nutzt für die ankommende Information dasselbe temporäre Matrixschema, er schreibt aber spaltenweise und liest anschließend zeilenweise aus, erhält also damit wieder m Wörter der Länge n.

Anschließend wird mit einem für unseren Code geeigneten Verfahren decodiert. Benötigen wir für unsere Information mehr als m Codewörter, was die Regel sein wird, so wiederholen wir das Verfahren mit den nächsten m Codewörtern.

Was haben wir damit erreicht? Treten bei der Übertragung Fehlerbündel auf oder sind ganze Teile des Datenträgers nicht lesbar, dann betrifft dies mehrere hintereinanderliegende Spaltenvektoren der temporären Matrix. Sagen wir also, s ganze Spaltenwörter hintereinander sind beim Empfang oder Lesen des Datenträgers extrem fehlerbehaftet oder gar ganz ausgelöscht, so beeinflusst dies aber *nur* s Positionen je Zeilenwort, also je Codewort. Ist der Code gut genug, dann kann er das korrigieren.

Beim Empfangen oder Auslesen der Information muss eine zeitliche Verzögerung wegen des Aufbaus der temporären Matrizen in Kauf genommen werden. Denn bevor nicht alle n Spalten in der Matrix eingetroffen sind, kann man ja die Zeilen noch nicht auslesen. Diesen Nachteil verringert das sog. **verzögerte Block-Interleaving**. Man verändert dabei die Ablageform ein wenig und legt die Codewörter aus C nicht als Zeile einer Matrix ab,

sondern *treppenförmig* nach rechts unten.

$$\begin{pmatrix} c_{11} & c_{21} & \dots & c_{i1} & c_{(i+1)1} & \dots & & & & \\ 0 & c_{12} & \dots & \dots & c_{i2} & c_{(i+1)2} & \dots & & & \\ 0 & 0 & c_{13} & \dots & \dots & c_{i3} & c_{(i+1)3} & \dots & & \\ \vdots & \vdots & \vdots & & & \vdots & & & & \\ 0 & \dots & \dots & \dots & \dots & \dots & \dots & c_{in} & c_{(i+1)n} & \dots \end{pmatrix}$$

Am Anfang wird mit 0 aufgefüllt. Man erhält also ein Schema der Tiefe n, bei dem durch jedes neue Codewort eine neue Spalte erzeugt wird. Man sendet dann aber wieder spaltenweise, also mit Wörtern, die die gleiche Länge n wie der Code haben. Wird dann ein solcher Spaltenvektor empfangen oder vom Datenträger ausgelesen, so vervollständigt sich mit dessen letzter Position auch gleichzeitig ein weiteres Codewort, mit dem dann sofort weitergearbeitet werden kann. Statt mit der eben beschriebenen einfachen Verzögerung kann man auch mit mehrfacher Verzögerung arbeiten. Hier schematisch das zweifach verzögerte Blockinterleaving.

$$\begin{pmatrix} \dots & c_{i1} & c_{(i+1)1} & \dots & \dots & \dots \\ \dots & \dots & \dots & c_{i2} & c_{(i+1)2} & \dots \end{pmatrix}$$

Wenn schon nicht die ursprünglichen Codewörter eines Codes C gesendet oder abgespeichert werden, sondern eher *künstlich* erzeugte Spaltenvektoren eines Matrix- oder Rautenschemas, dann kann man auch diese Vektoren einem weiteren Code C' unterwerfen und dann erst auf die Reise schicken oder eben abspeichern. Damit erhöht man das Fehlerkorrekturverhalten des Gesamtkonstrukts ein weiteres Mal. Man nennt dies dann **Cross-Interleaving** des **inneren Codes** C' mit dem **äußeren Code** C. Wenn es sich bei beiden Codes um Reed-Solomon-Codes handelt, spricht man auch kurz von **CIRC** (**Cross-Interleaved Reed-Solomon-Codes**). Natürlich muss der innere Code C' zu der Konstellation passen. Er sollte also über demselben Körper gebildet sein und als Dimension die Interleaving-Tiefe m haben. Wie so etwas in der Praxis aussieht, sehen wir im nächsten Abschnitt an dem Musterbeispiel *Kratzer* auf CD und DVD.

5.5 Audio-CDs und Daten-DVDs

5.5.1 Die Audio-CD – Technische Realisierung

Die CD ist ein optisches Massenspeichermedium. Die digitale Information ist dabei auf einer spiralförmig verlaufenden Spur in Form von Vertiefungen (sog. **Pits**) und Nichtvertiefungen (sog. **Lands**) abgespeichert. Die Übergänge Pit/Land bzw. umgekehrt markieren dabei eine 1, gefolgt von 0 auf den Pits bzw. Lands. Ein Bit entspricht dabei einer Länge von 0,3 µm. Abb. 5.6 visualisiert diesen Sachverhalt.

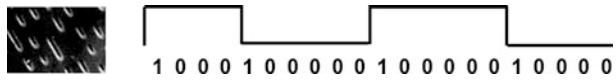


Abb. 5.6 CD unter Raster-Elektronen-Mikroskop (Quelle [WPCD]) sowie schematische Darstellung der Pits und Lands

Beim Abspielen wird die CD von einem Fotodetektor anhand der Intensität eines reflektierten Laserstrahls ausgelesen. Bei den Pits wird das Licht infolge von Interferenzen weniger stark reflektiert als bei den Lands. Die korrekte Lesbarkeit erfordert, dass zwischen zwei Einsen mindestens zwei Nullen stehen müssen, aus Synchronisationsgründen dürfen aber andererseits höchstens zehn Nullen stehen.

Das analoge Tonsignal wird 44.100-mal pro Sekunde (also mit 44,1 kHz) abgetastet. Die gewählte Abtastfrequenz hängt damit zusammen, dass sie zumindest das Doppelte der maximalen menschlichen Hörfrequenz sein muss. Jedem der abgetasteten Audiowerte wird auf einer Skala von 0 bis $2^{16} - 1 = 65.535$ ein numerischer Wert (sog. *Sample*) zugeordnet, der als binäre Zahl geschrieben wird, also als 16-Tupel über $\mathbb{Z}_2$. Dies macht man für jeden der beiden Stereokanäle.

5.5.2 Anwendung: CD-Brennen – Codierung auf der Audio-CD

Wir beschreiben jetzt Schritt für Schritt, wie die Speicherung auf CD erfolgt, so wie dies auch in Abb. 5.7 schematisch dargestellt ist.

- Bei jeder Messung des Musiksignals entsteht bei jedem der beiden Stereokanäle eine Bitfolge der Länge 16, also insgesamt der Länge 32. Fassen wir je 8 Bits (also 1 Byte) als binäre Darstellung im Körper K mit 2^8 Elementen auf, so liefern also die beiden Kanäle vier Elemente aus K. Je sechs Messungen werden zu einem Wort der Länge 24 über K zusammengefasst.
- Diese Wörter der Länge 24 werden mit dem – im letzten Abschnitt aus einem Reed-Solomon-Code abgeleiteten – MDS-Code mit den Parametern $[28, 24, 5]$ systematisch codiert. Er dient im Sinne des nun folgenden Cross-Interleavings als äußerer Code.

Abb. 5.7 Codierungsschema bei Audio-CDs

- Nun werden diese äußeren Codewörter der Länge 28 einem vierfach verzögerten Block-Interleaving von Tiefe 28 unterworfen.
- Im Anschluss werden die Spalten des Interleaving-Schemas mit dem aus einem Reed-Solomon-Code abgeleiteten MDS-Code mit den Parametern $[32, 28, 5]$ systematisch codiert. Dieser Code dient im Sinne des Cross-Interleavings als innerer Code. Damit ergeben sich also Wörter der Länge 32.
- In einem nächsten Schritt müssen aus technischen Gründen weitere Bits eingefügt werden. Wie oben beschrieben, müssen zwischen 2 Einsen mindestens 2 Nullen vorkommen (*Auflösung*), es dürfen aber höchstens 10 solche Nullen sein (*Synchronisation*). Dazu werden 8 Bits zu 14 Bits *aufgeblasen* (sog. 8-14-Modulation). Für die sechs Abtastungen sind wir bis jetzt auf eine Bitfolge der Länge $32 \cdot 14 = 448$ gekommen.
- Zuallerletzt werden noch weitere technische Pufferbits eingebaut, die wir hier aber nicht weiter beschreiben wollen. Damit landen wir am Ende bei unseren sechs Messungen bei insgesamt 588 Bits, die dann auch so auf die CD gebrannt werden.

5.5.3 Anwendung: Musikhören – Decodierung im CD-Spieler

Was im Einzelnen beim Abhören der CD passiert, erfahren wir jetzt.

- Beim Lesen der CD werden zunächst die oben beschriebenen technischen Bits vom Bitstrom entfernt.
- Danach kommt der innere Code zur Anwendung. Wegen seines Minimalgewichts von 5 könnte er genutzt werden, um bis zu zwei Fehler je Wort der Länge 32 zu korrigieren. Stattdessen wird aber wie folgt verfahren.
  - Gibt es ein Codewort mit Abstand maximal 1 vom gelesenen Wort, so wird zu diesem Codewort korrigiert.
  - Gibt es kein solches, so wird ein Fehler festgestellt und wie im nächsten Schritt beschrieben weiter verfahren. Man beachte, dass bei diesem Ansatz zwei oder drei Fehler sicher erkannt werden, aber nicht vier oder mehr Fehler, da hier ja möglicherweise zu einem anderen falschen Codewort mit Abstand 1 korrigiert wird.
  - Kurze Fehlerabschätzung zwischendurch: Wir nehmen dazu an, dass alle denkbaren Fehler mit gleicher Wahrscheinlichkeit auftreten. Dann beträgt die Wahrscheinlichkeit, dass ein Fehler nicht erkannt wird, weil das ausgelesene Wort in die Kugel vom Radius 1 um ein anderes Codewort geraten ist: $(q^k-1)(1+n(q-1))/q^n < nq/q^{n-k}$. Für die Parameter unseres Codes, nämlich $q = 2^8$, $k = 28$ und $n = 32$, ergibt dies $32q/q^4 = 2^{13}/2^{32} = 2^{-19} < 2 \cdot 10^{-6}$. Mit dem inneren Code erkennt man also beliebige Fehler mit Wahrscheinlichkeit $> 99{,}9998\,\%$.
- Stellt der Decodieralgorithmus des inneren Codes also ein fehlerhaftes Wort fest, das er nicht korrigieren kann, so stuft er alle 28 Positionen der Spalte im Interleaving-Schema, aus dem das Wort der Länge 32 stammt, als unlesbar ein, also als Auslöschung.

- Jetzt wird unser Interleaving wirksam. Nehmen wir dazu an, dass durch ein ganzes Fehlerbündel bis zu 16 aufeinanderfolgende Spalten des Interleaving-Schemas ausgelöscht werden. Im vierfach verzögerten Interleaving-Schema sind aber die Codewörter des äußeren Codes treppenförmig mit Stufen der Länge 4 angeordnet. Daher werden je Codewort im äußeren Code nur maximal vier Positionen von diesen 16 ausgelöschten Spalten betroffen. Wie wir aber bereits in Abschn. 1.3 gesehen haben, kann der äußere Code mit seinem Minimalabstand 5 bis zu vier Auslöschungen korrigieren.

Insgesamt sind auf diese Weise $16 \cdot 32 \cdot 8 = 4096$ Bits rekonstruierbar. Wegen der oben erwähnten zusätzlichen technischen Bits sind diese 4096 Bits in genau $16 \cdot 588 = 9408$ Bits auf der CD enthalten. Damit kann der Audioinhalt, der auf einer Spurlänge von annähernd 3 mm abgespeichert ist, komplett rekonstruiert werden. Bei Audio-CDs kann man die Korrekturfähigkeit noch weiter erhöhen, indem man das Mittel der **Interpolation** einsetzt. Wenn nämlich Daten ausgelöscht wurden, so kann man deren Werte mittels nichtausgelöschter oder über Codierverfahren rekonstruierter Stützwerte wiederherstellen – zumindest innerhalb einer akzeptablen Toleranz bzgl. menschlicher Hörfähigkeit. Auf diese Art erreicht man eine rekonstruierbare Spurlänge von bis zu 7,5 mm.

5.5.4 Anwendung: Datencodierung auf DVD

Die **DVD**, **Digital Versatile Disc** (dt. digitale vielseitige Scheibe), ähnelt im Aussehen einer CD, verfügt aber über eine deutlich höhere Speicherkapazität. Sie zählt ebenfalls zu den optischen Datenspeichern. Im Vergleich zu den CDs wird bei DVDs mit Lasern kürzerer Wellenlänge gearbeitet, mit denen in den Datenträgerschichten entsprechend kleinere Strukturen gelesen und geschrieben werden können.

Kommen wir gleich wieder zur Vorgehensweise bei der Fehlerkorrektur. Wie bei der Audio-CD wird bei Daten-DVDs mit Cross-Interleaving gearbeitet, allerdings mit unverzögertem Interleaver. Außerdem werden mit fortgeschrittener Technik auch leistungsfähigere Reed-Solomon-Codes eingesetzt, nämlich die im letzten Abschnitt abgeleiteten MDS-Codes mit den Parametern $[182, 172, 11]$ und $[208, 192, 17]$ über dem Körper K mit 2^8 Elementen. Jedes Element aus K wird dabei wieder als binäres 8-Tupel (Byte) aufgefasst. Die Informationsinhalte der DVD werden zu Wörtern der Länge 172 über K zusammengefasst und mittels des ersten äußeren Codes in Codewörter der Länge 182 umgewandelt. Von diesen werden 192 Stück zeilenweise in eine temporäre Matrix geschrieben, welche also genau die beiden oberen Teilmatrizen in Abb. 5.8 umfasst. Dann wird der zweite innere Code auf die Spalten dieser Matrix angewandt und aus den Spalten der Länge 192 werden Spalten der Länge 208.

Im normalen Cross-Interleaving würden jetzt die Spalten dieser Matrix gesendet bzw. auf DVD gespeichert. Hier wird allerdings vorher noch eine kleine Veränderung vorgenommen. Es werden die letzten 16 Zeilen mit den Paritäts-Zeichen hinter jede zwölfte Zeile der Teilmatrix darüber verschoben. Formal liest sich diese Permutation der $c_{i,j}$ in

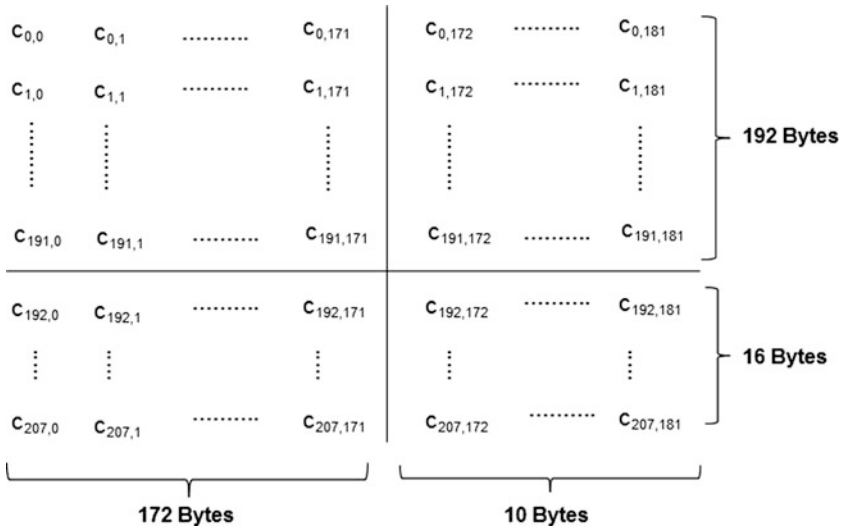


Abb. 5.8 Interleaving-Schema bei DVDs

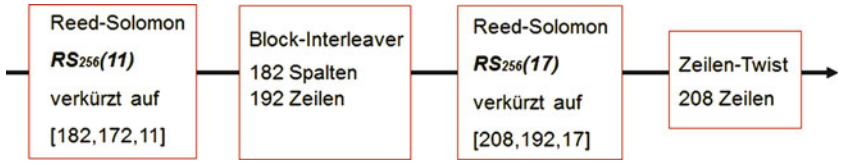


Abb. 5.9 Codierungsschema bei Daten-DVDs

$c_{m,n}$ wie folgt:

$$m = \begin{cases} i + \lfloor i/12 \rfloor & \text{für } i < 192 \ (\lfloor\rfloor = \text{ ganzzahliger Anteil}); \\ 13(i - 191) - 1 & \text{für } i \geq 192; \end{cases}$$
$$n = j.$$

Die so entstandenen Spalten werden jetzt auf DVD geschrieben. Ist die Matrix vollständig ausgelesen, dann werden die nächsten Informationseinheiten der Länge 172 codiert und in die temporäre Matrix eingelesen. Auch hier ist das gesamte Codierungsschema für DVDs in Abb. 5.9 zusammengefasst.

Das Decodierverfahren funktioniert ähnlich wie bei der CD. Interpolation ist bei Daten-DVDs aber leider nicht möglich.

5.5.5 Anwendung: Festplatten

Auch bei anderen Massenspeichermedien, wie vor allem **Festplatten**, werden Reed-Solomon-Codes schon seit langer Zeit zur Datencodierung eingesetzt. Allerdings haben in jüngerer Vergangenheit dort auch andere Codes Einzug gehalten – sog. **LDPC-Codes**. Aus diesem Grund gehen wir auf Festplattenlaufwerke erst in Abschn. 8.1 ein.

Zyklische Codes und CRC-Verfahren

6

Grundlagen zyklischer Codes Bislang haben wir Polynome ausgewertet und so Reed-Solomon-Codes konstruiert. Jetzt multiplizieren wir Polynome zur Abwechslung mal. Wir stellen uns dazu das n-Tupel der Codewörter als Koeffizienten von Polynomen vor und nennen einen Code **zyklisch**, wenn alle Codewörter ein Polynomvielfaches eines festen Polynoms $g(x)$ sind. Das Polynom $g(x)$ nennt man dann **Generatorpolynom**. Ein Vorteil von zyklischen Codes ist unmittelbar klar: Bei linearen Codes mussten wir uns die gesamte Generatormatrix merken, bei zyklischen Codes genügt es, das Generatorpolynom zu kennen. Wir fangen mit kleinen Beispielen zum Eingewöhnen an. Das ist allerdings gar nicht so schwer, sodass wir uns gleich davon überzeugen können, dass auch die beiden perfekten **Golay-Codes zyklische Codes** sind.

Schieberegister Das Konzept der zyklischen Codes scheint auch zu tragen. Um mit ihnen effizient rechnen zu können, muss man die **Polynommultiplikation** und **Polynomdivision** (ggf. **mit Rest**) auf Basis schneller digitaler Schaltungen implementieren. Das Mittel der Wahl sind nach wie vor sog. **Schieberegister**, deren Funktionsweise wir untersuchen. Die Effizienz von Schieberegisterschaltungen ist ein wesentlicher Grund dafür, dass zyklische Codes bevorzugt Anwendung finden. Schieberegister werden uns aber auch noch später beim Berlekamp-Massey-Algorithmus und bei den Faltungscodes begegnen.

Codierung zyklischer Codes Fangen wir also mal einfach an. Wir **codieren** nämlich zuerst einen **zyklischen Code**, indem wir mittels Schieberegisterschaltung mit seinem Generatorpolynom multiplizieren. Soweit so gut, aber es bleibt ein Problem: Am liebsten codiert man ja **systematisch**, wobei dann die Information an den ersten k Koeffizienten des Codepolynoms ablesbar ist. Wir sehen, dass man dies mittels Polynomdivision durch das Generatorpolynom bewerkstelligen kann, und tun dies auch ganz explizit mit einem Schieberegister anhand des zweitkleinsten binären Hamming-Codes.

O. Manz, *Fehlerkorrigierende Codes*, DOI 10.1007/978-3-658-14652-8_6

Meggitt-Decodierung Jetzt wieder zum leidigen – weil schwierigeren – Thema Decodieren. Wir lernen zuerst das Standarddecodierverfahren für zyklische Codes kennen, welches mit sog. **Syndrompolynomen** arbeitet. Der Ansatz entspricht dem der Syndromdecodierung linearer Codes. Unter **Meggitt-Decodierung** versteht man eine kleine Modifikation des Verfahrens, welche allerdings die Laufzeit deutlich verbessert. Wir nutzen auch hier den zweitkleinsten binären Hamming-Code für ein explizites Decodierbeispiel.

CRC – Cyclic Redundancy Check Bislang haben wir bei unseren Codes eher an Fehlerkorrektur (**FEC, Forward Error Correction**) gedacht, als an die bloße Fehlererkennung. Diese spielt aber unter dem Stichwort **ARQ (Automatic Repeat Request)** eine wichtige Rolle insbesondere bei Computernetzwerken. Dabei hängt man an ein Datenpaket, d. h. an eine Bitfolge, ein paar Bits als **FCS (Frame Checking Sequence)** an. Das einfachste Beispiel einer solchen FCS kennen wir schon, nämlich ein Paritätsprüfungsbit. Der Empfänger prüft auf Korrektheit und fordert – falls inkorrekt – eine Neuversendung des Datenpakets an. Das alles muss sicher sein und vor allem schnell gehen. Daher sind zyklische Codes das Mittel der Wahl, man nennt das FCS-Verfahren dann **CRC (Cyclic Redundancy Check)**. Es geht auf **William Wesley Peterson** und das Jahr 1961 zurück. Man verwendet dabei die gleichen Schieberegisterschaltungen, um mittels Division mit Rest systematisch zu codieren und auf Fehler zu überprüfen. Wir lernen die wichtigsten standardisierten **CRC-Polynome** kennen sowie deren Fehlererkennungseigenschaften.

Computernetzwerke, Schnittstellen und Speicherchips Die Liste der Anwendungen von CRC ist lang. Das CRC-Polynom **CRC-16-CCITT** beispielsweise wird verwendet bei der drahtlosen **Bluetooth**-Schnittstelle sowie bei der **SD-Card** von SanDisk. Das **CRC-16-IBM**-Verfahren findet Anwendung beim Protokoll **Modbus** in der Prozessleittechnik sowie beim universellen Schnittstellenstandard **USB**. Wichtige Anwendungen von **CRC-32** sind das **Ethernet** und das **WLAN**. Eher enttäuschend ist das FCS-Verfahren, welches im **Internet TCP/IP** verwendet wird. Dahingegen verleiht **CRC-24-Q** der **PGP-Verschlüsselung** (**Pretty Good Privacy**) und damit dem E-Mail-Verkehr im Internet eine deutlich höhere Fehlererkennungsstufe. Wir schauen uns all diese Anwendungen genauer an.

6.1 Grundlagen zyklischer Codes

6.1.1 Zwischenstatus und *Polynomcodewörter*

Lehnen wir uns also zunächst etwas zurück und lassen Revue passieren, was wir bis jetzt gemacht haben.

- Zunächst haben wir uns überlegt, dass man gute fehlererkennende oder -korrigierende Codes C erhält, wenn man zu einem Alphabet A nach Teilmengen $C \subseteq A^n$ sucht, die neben geeigneter Mächtigkeit $|C|$ auch möglichst großen Minimalabstand besitzen.

Dabei muss man sich aber alle Codewörter der Teilmenge C merken und bei Hamming-Decodierung alle durchsuchen.

- Dies hat uns auf die Idee gebracht, für das Alphabet A kleine endliche Körper K (in der Regel $\mathbb{Z}_2$ oder $\mathbb{Z}_3$) zu verwenden und nur noch lineare Codes C zu betrachten, d. h. Unterräume des K^n. Dann nämlich reicht es aus, sich eine Basis von C zu merken, genau genommen die Generatormatrix, die ja als Zeilen eine Basis von C enthält.
- Als Nächstes haben wir auf K^n auch das Skalarprodukt $\langle\ ,\ \rangle$ untersucht. Dies führte u. a. auf die Kontrollmatrix, d. h. auf die Generatormatrix des dualen Codes, auf Kontrollgleichungen und damit auf die Syndrom- und Majority-Logic-Decodierung.
- Nachdem wir mehrere Serien von guten linearen Codes konstruiert und untersucht hatten, stellte sich heraus, dass wir mit kleinen Körpern nicht mehr weiterkommen. Wir benötigten insbesondere die Körper K mit 2^m Elementen, die wir mittels Polynome konstruiert haben.
- Polynome konnten wir auch verwenden, um Auswertungscodes zu definieren, d. h. die Positionen der Codewörter bestehen aus Polynomauswertungen. Dies ergab die sehr guten Reed-Solomon-Codes.
- Wie sind Polynome sonst noch verwendbar? Nun, bei linearen Codes, d. h. Unterräumen in K^n, können wir *nur* Linearkombinationen von Vektoren bilden. Polynome hingegen kann man zusätzlich noch multiplizieren und ggf. mit Rest dividieren. Dieser Ansatz würde es uns ermöglichen, noch mehr algebraische Struktur in die Codierungstheorie einzubringen.

Man kann sich die Koeffizienten eines Polynoms $f(x) = a_{n-1}x^{n-1} + \ldots + a_1x + a_0$ auch als n-Tupel $(a_{n-1}, \ldots, a_0) \in K^n$ vorstellen – oder auch in umgekehrter Reihenfolge $(a_0, \ldots, a_{n-1}) \in K^n$, wenn man die Summe der x-Potenzen im Polynom lieber aufsteigend schreibt. Also können wir die Polynom- und Tupelbetrachtungsweisen identifizieren und uns lineare Codes der Länge n auch als Teilmenge $C \subseteq K[x]_{n-1}$ des Vektorraumes der Polynome vom Grad $\leq n-1$ vorstellen.

6.1.2 Was versteht man unter zyklischen Codes?

Ein linearer Code $C \subseteq K[x]_{n-1}$ über K der Länge n nennt man **zyklisch**, wenn für alle $c(x) \in C$ und alle Polynome $f(x) \in K[x]$ deren Produkt modulo des Polynoms $x^n - 1$ wieder ein Codewort in C ist, d. h. wenn $c(x)f(x) \in C \pmod{x^n - 1}$ gilt.

Man beachte, dass der Rest modulo $x^n - 1$ wieder in $K[x]_{n-1}$ liegt, also auch wieder Länge n hat. Algebraisch gesehen nennt man so eine Menge **Ideal**.

Woher kommt der Begriff zyklisch? Sei dazu $a_{n-1}x^{n-1} + \ldots + a_1x + a_0 \in C$, in Tupelschreibweise heißt das $(a_{n-1}, \ldots, a_0) \in C$. Die Multiplikation mit x und Division durch $x^n - 1$ mit Rest liefert $(a_{n-1}x^{n-1} + \ldots + a_1x + a_0)x = a_{n-1}x^n + \ldots + a_1x^2 + a_0x = a_{n-2}x^{n-1} + \ldots + a_1x^2 + a_0x + a_{n-1} \in C (\mathrm{mod}\, x^n - 1)$. In Tupelschreibweise heißt das $(a_{n-2}, \ldots, a_1, a_0, a_{n-1}) \in C$, also bleibt C als Menge erhalten bei **zyklischer** Vertauschung der Positionen.

6.1.3 Generator- und Kontrollpolynom

Wir sammeln nun ein paar Eigenschaften von zyklischen Codes und üben bei dieser Gelegenheit das Rechnen mit Polynomen.

Sei C ein zyklischer Code über K der Länge n und Dimension k. Wir wählen $0 \neq g(x) \in C$ normiert (d. h. höchster Koeffizient $= 1$) und von minimalem Grad. Dann lässt sich $x^n - 1$ schreiben als $x^n - 1 = g(x)h(x)$ und die beiden Polynomen $g(x)$ und $h(x)$ sind eindeutig bestimmt. Dabei heißt $g(x)$ **Generatorpolynom** und $h(x)$ **Kontrollpolynom** zu C.

Wir überlegen uns das und dividieren hierzu $x^n - 1$ mit Rest durch $g(x)$, also $x^n - 1 = g(x)h(x) + r(x)$ mit $h(x), r(x) \in K[x]$ und $\mathrm{grad}(r(x)) < \mathrm{grad}(g(x))$. Wir folgern $g(x)h(x) = -r(x) (\mathrm{mod}\, x^n - 1)$, also $r(x) \in C$, da $g(x) \in C$ und das C ein zyklischer Code ist. Die Minimalität von $\mathrm{grad}(g(x))$ erzwingt $r(x) = 0$, also $x^n - 1 = g(x)h(x)$. Sei auch $f(x) \in C$ normiert und von minimalem Grad, so ist $\mathrm{grad}(g(x)) = \mathrm{grad}(f(x))$ und folglich $\mathrm{grad}(g(x) - f(x)) < \mathrm{grad}(g(x))$. Da andererseits $g(x) - f(x) \in C$ ist, folgt $g(x) - f(x) = 0$ und $g(x)$ ist eindeutig bestimmt. Auch $h(x)$ ist dann eindeutig als Divisionsergebnis von $x^n - 1$ durch $g(x)$.

Sei $g(x)$ das Generatorpolynom zu C vom Grad m. Dann ist $C = \{g(x)f(x) | f(x) \in K[x]_{n-1-m}\}$ und $\{g(x), g(x)x, g(x)x^2, \ldots, g(x)x^{n-1-m}\}$ ist eine Basis von C. Insbesondere gilt $k = n - m$.

Auch das wollen wir uns überlegen. Wegen $\mathrm{grad}(g(x)x^i) = m + i$ sind die $g(x)x^i$ jedenfalls mal linear unabhängig. Wir müssen noch zeigen, dass sie ganz C als Vektorraum erzeugen. Sei also $c(x) \in C$, insbesondere $\mathrm{grad}(c(x)) \leq n - 1$. Wir dividieren c(x) durch $g(x)$ mit Rest, also $c(x) = g(x)f(x) + r(x)$ mit $\mathrm{grad}(r(x)) < m \leq n - 1$. Also hat $g(x)f(x) = c(x) - r(x)$ ebenfalls Grad $\leq n-1$ und liegt damit wieder in C, da $g(x) \in C$ und C zyklisch ist. Damit ist aber auch $r(x) = c(x) - g(x)f(x) \in C$ und die Minimalität

von grad$(g(x))$ erzwingt $r(x) = 0$, also $c(x) = g(x)f(x)$. Dabei hat $f(x)$ höchstens den Grad $n-1-m$ und wir können $f(x)$ schreiben als $f(x) = a_0+a_1x+\ldots+a_{n-1-m}x^{n-1-m}$ für geeignete $a_i \in K$. Somit folgt $c(x) = a_o g(x) + a_1 g(x)x + +a_{n-1-m}g(x)x^{n-1-m}$.

Wir haben einen zyklischen Code C als Teilmenge von $K[x]_{n-1}$ definiert und eben mittels seines Generatorpolynoms $g(x)$ eine Darstellung und eine Basis hergeleitet, bei der alle Polynome ebenfalls in $K[x]_{n-1}$ liegen. Man muss aber immer noch *im Hinterkopf* haben, dass man ein Codewort $c(x) \in C$ auch mit einem Polynom $f(x)$ beliebigen Grads multiplizieren darf, man dabei das Ergebnis aber als Rest modulo $x^n - 1$ lesen muss, welcher dann auch wieder in C liegt.

6.1.4 Generator- und Kontrollmatrix zyklischer Codes

Bei zyklischen Codes genügt es also, sich anstelle der Generatormatrix nur das Generatorpolynom zu merken. Wir wollen dennoch Generator- und Kontrollmatrizen zyklischer Codes konkret notieren. Sei dazu C ein zyklischer $[n,k]$-Code über K mit Generatorpolynom $g(x) = g_0 + \ldots + g_{n-k}x^{n-k}$ und Kontrollpolynom $h(x) = h_0 + \ldots + h_k x^k$.

Die folgende Matrix G mit n Spalten und k Zeilen ist Generatormatrix von C.

$$G = \begin{pmatrix} g_0 & g_1 & \ldots & g_{n-k} & 0 & \ldots & 0 \\ 0 & g_0 & g_1 & \ldots & g_{n-k} & \ldots & 0 \\ \vdots & \vdots & \vdots & \ldots & \vdots & \ldots & \vdots \\ 0 & 0 & \ldots. & g_0 & g_1 & \ldots & g_{n-k} \end{pmatrix}$$

Die folgende Matrix H mit n Spalten und n - k Zeilen ist Kontrollmatrix von C.

$$H = \begin{pmatrix} h_k & h_{k-1} & \ldots & h_0 & 0 & \ldots & 0 \\ 0 & h_k & h_{k-1} & \ldots & h_0 & \ldots & 0 \\ \vdots & \vdots & \vdots & \ldots & \vdots & \ldots & \vdots \\ 0 & 0 & \ldots & h_k & h_{k-1} & \ldots & h_0 \end{pmatrix}$$

Um G als Generatormatrix zu identifizieren, muss man nur die Koeffizienten der Basisvektoren $\{g(x), g(x)x, g(x)x^2, \ldots, g(x)x^{k-1}\}$ von C zeilenweise in die Matrix schreiben. Bei der Kontrollmatrix muss man sich erinnern, dass die Zeilenvektoren von H eine Basis von $C^\perp$ sind, dass also die Zeilen von G und die Zeilen von H jeweils Skalarprodukt $\langle\ ,\ \rangle = 0$ miteinander haben und die Zeilen von H linear unabhängig sein müssen. Genau dies aber liest man aus der Gleichung $g(x)h(x) = x^n - 1$ ab. Durch distributives Ausmultiplizieren der linken Seite erhält man nämlich $g(x)h(x) = \sum_i (\sum_j g_j h_{i-j})x^i$, wobei der Index i von 0 bis n und der Index j von 0 bis i läuft und wobei alle nicht definierten g_i und h_j gleich 0 zu setzen sind. Der Koeffizientenvergleich ergibt $\sum_j g_j h_{i-j} = 0$ für alle $i > 0$ und $g_0 h_0 = -1$. Für $i > 0$ sind das genau die Skalarprodukte der Zeilen von G mit

den Zeilen von H. Aus $g_0 h_0 = -1$ folgt andererseits $h_0 \neq 0$ und folglich sind die Zeilen von H auch linear unabhängig.

6.1.5 Beispiel: Zyklische Codes

Um also alle zyklischen Codes einer vorgegebenen Länge n zu konstruieren, müssen wir uns deren Generatorpolynome und damit sämtliche Teiler von $x^n - 1$ verschaffen, was allerdings für große n nicht immer ganz leicht ist.

Binäre zyklische Codes der Länge 4
Fangen wir also mal klein an und suchen alle binären zyklischen Codes C der Länge $n = 4$ über $\mathbb{Z}_2$: Dazu müssen wir alle Teiler von $x^4 - 1 = x^4 + 1 = (x+1)^4$ finden. Hier sind sie, mit aufsteigendem Grad.

$g(x) = 1$. Dann ist $\{1, x, x^2, x^3\}$ eine Basis von C und damit $C = \mathbb{Z}_2^4$.

$g(x) = x + 1$. Dann ist $\{x+1, x^2+x, x^3+x^2\}$ eine Basis von C und die Generatormatrix lautet:

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

Es handelt sich also um einen Paritätsprüfungscode, wie man durch Überführung in systematische Form sofort sieht.

$g(x) = (x+1)^2 = x^2 + 1$. Dann ist $\{x^2+1, x^3+x\}$ eine Basis von C und die Generatormatrix lautet:

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

Hierbei handelt es sich also um einen Wiederholungscode.

$g(x) = (x+1)^3 = x^3 + x^2 + x + 1$. Jetzt ist C eindimensional mit Basis $\{g(x)\}$ und Generatormatrix

$$G = \begin{pmatrix} 1 & 1 & 1 & 1 \end{pmatrix}.$$

$g(x) = (x+1)^4 = x^4 + 1$. Dies führt zu $C = \{0\}$.

Der kleine binäre Hamming-Code $Ham_2(3)$ als zyklischer Code
Jetzt werden wir mutiger und betrachten binäre zyklische Codes der Länge $n = 7$ über $\mathbb{Z}_2$: Das Polynom $x^7 - 1 = x^7 + 1$ lässt sich faktorisieren als $(x+1)(x^3+x^2+1)(x^3+x+1)$ und wir wählen als Generatorpolynom $g(x) = x^3 + x + 1$. Dann ist $\{x^3 + x + 1, x^4 +$

$x^2 + x, x^5 + x^3 + x^2, x^6 + x^4 + x^3\}$ eine Basis unseres gesuchten Codes und

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

ist eine Generatormatrix. Wir bestimmen jetzt auch noch die Kontrollmatrix H. Das Kontrollpolynom $h(x)$ lautet jedenfalls $h(x) = (x + 1)(x^3 + x^2 + 1) = x^4 + x^2 + x + 1$. Also folgt

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

Wenn wir uns die Spalten von H anschauen, stellen wir fest, dass dies alle binären 3-Tupel ungleich $(0, 0, 0)$ sind. Wir folgern daraus, dass es sich um eine äquivalente Form unseres wohlvertrauten **kleinen binären Hamming-Codes** $Ham_2(3)$ handelt und dass dieser also zyklisch ist. Auf andere Hamming-Codes kommen wir in Abschn. 7.1 noch zurück.

Der ternärer Golay-Code G_{11} als zyklischer Code

Jetzt sollten wir doch auch mal einen Blick auf $n = 11$ und $K = \mathbb{Z}_3$ werfen. Wie schon gesagt, bei größeren n sieht man die Faktoren von $x^n - 1$ nicht so auf einen Blick. Aber durch Nachrechnen bestätigt man $x^{11} - 1 = (x - 1)(x^5 - x^3 + x^2 - x - 1)(x^5 + x^4 - x^3 + x^2 - 1)$. Wir wählen als Generatorpolynom $g(x) = x^5 - x^3 + x^2 - x - 1$. Dann lautet die zugehörige Generatormatrix

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

Man kann sich überlegen – was einigen Rechenaufwand mit den Spalten und Zeilen der Matrix bedarf und wir hier nicht explizit tun wollen –, dass dieser zyklische Code äquivalent zum perfekten **ternären Golay-Code** G_{11} ist. Also ist auch G_{11} zyklisch und wir haben hiermit – wie angekündigt – ein weiteres Konstruktionsprinzip von G_{11} kennengelernt. Man hätte übrigens auch den anderen der beiden Faktoren $x^5 + x^4 - x^3 + x^2 - 1$ als Generatorpolynom wählen können. Dies führt ebenfalls auf eine äquivalente Form von G_{11}.

Der binäre Golay-Code G_{23} als zyklischer Code
Jetzt kommt das, was wohl jeder erwartet. Wir nehmen $n = 23$ und $K = \mathbb{Z}_2$. Dann lässt sich $x^{23}+1$ faktorisieren in $x^{23}+1 = (x+1)(x^{11}+x^9+x^7+x^6+x^5+x+1)(x^{11}+x^{10}+x^6+x^5+x^4+x^2+1)$. Wählt man als Generatorpolynom einen der beiden Faktoren vom Grad 11, so erhält man jeweils 12-dimensionale Codes, deren Generatormatrizen wir nicht hinschreiben wollen und dies gerne dem Leser als Übung überlassen. Jedenfalls ergeben beide Generatorpolynome Codes, die äquivalent zum perfekten **binären Golay-Code** G_{23} sind. Also ist auch G_{23} zyklisch und wir haben gleichzeitig ein weiteres – möglicherweise überschaubareres – Konstruktionsprinzip für G_{23} kennengelernt.

6.1.6 Duale Codes zyklischer Codes

Ist vielleicht das Kontrollpolynom h(x) eines zyklischen Codes C der Dimension k auch das Generatorpolynom des dualen Codes $C^\perp$? Leider nicht ganz, denn in der Kontrollmatrix stehen die Koeffizienten h_j in umgekehrter Reihenfolge als die g_i in der Generatormatrix. Dies kann man im Kontrollpolynom von C aber einfach korrigieren, nämlich durch $x^k h(1/x)$. Dieses Polynom ist aber noch nicht normiert. Also müssen wir es noch normieren, nämlich $h_0^{-1} x^k h(1/x)$. Das also ist das Generatorpolynom von $C^\perp$ und duale Codes von zyklischen Codes sind somit ebenfalls zyklisch.

Leider kann man bei zyklischen Codes keine allgemeine Aussage über deren Minimalabstand treffen. Jetzt aber erst mal zum vielleicht wichtigsten Grund, warum man gerne mit zyklischen Codes arbeitet: Codierung und Decodierung lassen sich schnell und effizient implementieren.

6.2 Schieberegister

In diesem Abschnitt soll es um **Schieberegister** gehen. Schieberegister benötigen wir einerseits für die Codierung und Decodierung zyklischer Codes, die wir in den nächsten beiden Abschnitten besprechen werden, aber auch später, wenn es um den Berlekamp-Massey-Algorithmus (Abschn. 7.5) und um Faltungscodes (Abschn. 9.1) gehen wird.

6.2.1 Aufbau von Schieberegistern

Ein **Schieberegister** umfasst eine Kette von Speicherzellen, die in der Regel von links nach rechts angeordnet ist. Eine Zelle im Schieberegister nennt man auch **Zustand**. Jeder Zustand enthält ein Körperelement. Zu festen Zeitintervallen – **Takt** genannt – werden die Körperelemente in den einzelnen Zuständen einen Zustand weiter *von links nach rechts geschoben* (in Millionstel Sekundenbruchteilen).

Mittels Schieberegister kann man sehr effizient in endlichen Körpern K und mit Polynomen $K[x]$ über K rechnen. Die Konvention, Schieberegister von links nach rechts zu

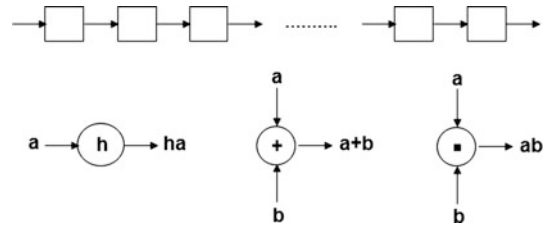


Abb. 6.1 Schieberegisterzellen und -schaltelemente

lesen, hat sich eingebürgert. Wenn man dabei Polynomkoeffizienten in die Zellen schreibt, beginnt man in der Regel mit dem höchsten, der dann ganz rechts steht.

Außerdem enthält ein Schieberregister noch bis zu drei weitere Schaltelemente, deren Symbolik in Abb. 6.1 gezeigt wird.

- Skalierer multipliziert ein Körperelement mit einem festen Körperelement.
- Addierer addiert zwei Körperelemente.
- Multiplizierer multipliziert zwei Körperelemente.

6.2.2 Polynommultiplikation mittels Schieberegister

Um das Ganze nicht zu theoretisch werden zu lassen, betrachten wir sofort das Beispiel einer Polynommultiplikation. Sei dazu $g(x) = g_L x^L + \ldots + g_1 x + g_0$ ein festes Polynom. Wir wollen ein weiteres Polynom $a(x) = a_k x^k + \ldots + a_1 x + a_0$, welches wir als Eingabe (**Input**) für das Schieberegister verwenden, mit $g(x)$ multiplizieren. Es soll also $g(x)a(x) = b(x) = b_{k+L} x^{k+L} + \ldots + b_1 x + b_0$ berechnet und das Polynom $b(x)$ dann am Ende vom Schieberegister herausgegeben werden (**Output**). Das Produktpolynom $b(x)$ hat als Koeffizienten $b_i = \sum_j g_j a_{i-j}$, wobei die Summe über j von 0 bis i läuft und alle nicht im Polynom $a(x)$ oder $g(x)$ definierten Koeffizienten a_i, g_j als 0 zu interpretieren sind. Wie diese Multiplikation im Schieberegister abläuft, machen wir uns anhand von Abb. 6.2 klar.

- Am Anfang sind alle Zustände (Zellen) mit 0 initialisiert.
- Danach werden der Reihenfolge nach $a_k, a_{k-1}, \ldots, a_0$ bei jedem Takt von links nach rechts in die Zellen geschoben.
- Bei jedem solchen Takt wird ein b_j entsprechend der obigen Bildungsregel erzeugt und rechts ausgegeben.
- Nach Input von a_0 wird dann bei jedem Takt mit 0 aufgefüllt, bis alle Zustände wieder 0 sind.
- Unser gewünschtes Ergebnis $b_{L+k}, \ldots, b_0$ verlässt das Schieberegister in dieser Reihenfolge nach rechts.

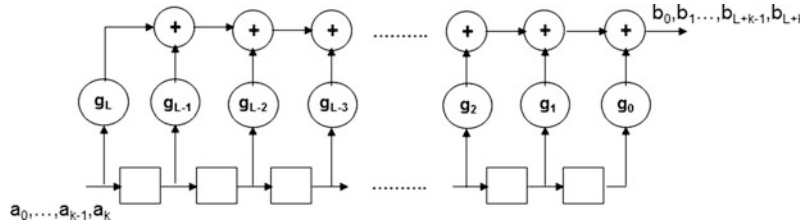


Abb. 6.2 Schieberegister zur Polynommultiplikation

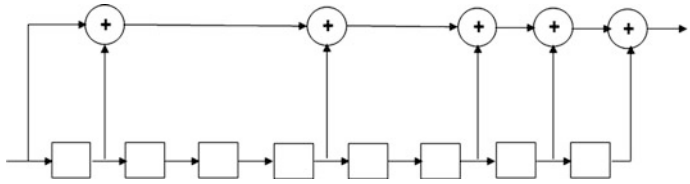


Abb. 6.3 Polynommultiplikation mit $g(x) = x^8 + x^7 + x^4 + x^2 + x + 1$

6.2.3 Beispiel: Polynommultiplikation mittels Schieberegister

Zur Erläuterung jetzt gleich ein Beispiel eines Schieberegisters für ein konkretes Polynom, nämlich $g(x) = x^8+x^7+x^4+x^2+x+1 \in \mathbb{Z}_2[x]$. Bei binären Polynomen benötigen wir die Skalierer im Schieberegister nicht, denn entweder wird direkt verschaltet (bei $g_j = 1$) oder eben gar nicht (bei $g_j = 0$). Abb. 6.3 zeigt das Schieberegister für die Multiplikation mit $g(x)$.

Wir wollen zur Übung das Polynom $a(x) = x^2 + 1$ mit $g(x)$ multiplizieren. Dies liest sich dann bzgl. Input-, Zellen- und Outputwerte wie folgt:

Takt	Input	Zellen	Output
Init	1 0 1	0 0 0 0 0 0 0 0	
1	1 0	1 0 0 0 0 0 0 0	1
2	1	0 1 0 0 0 0 0 0	1 1
3		1 0 1 0 0 0 0 0	1 1 1
4	1. 0-Input	0 1 0 1 0 0 0 0	1 1 1 1
5	2. 0-Input	0 0 1 0 1 0 0 0	1 1 1 1 1
6	3. 0-Input	0 0 0 1 0 1 0 0	0 1 1 1 1 1
7	4. 0-Input	0 0 0 0 1 0 1 0	0 0 1 1 1 1 1
8	5. 0-Input	0 0 0 0 0 1 0 1	1 0 0 1 1 1 1 1
9	6. 0-Input	0 0 0 0 0 0 1 0	0 1 0 0 1 1 1 1 1
10	7. 0-Input	0 0 0 0 0 0 0 1	1 0 1 0 0 1 1 1 1 1
11	8. 0-Input	0 0 0 0 0 0 0 0	1 1 0 1 0 0 1 1 1 1 1

Für das Produktpolynom $b(x)$ ergibt sich also $b(x) = x^{10}+x^9+x^8+x^7+x^6+x^3+x+1$, verbunden mit dem nochmaligen Hinweis, dass bei den Ausgabeparametern der höchste Koeffizient ganz rechts steht.

6.2.4 Polynomdivision mittels Schieberegister

Mit einem Schieberegister kann man aber auch unser Eingabepolynom a(x) durch das feste Polynom $g(x)$ dividieren, möglicherweise mit Rest. Dazu normieren wir $g(x)$ zuerst, sodass wir $g_L = 1$ voraussetzen können. Um die Funktion des Schieberegisters zu verstehen, dividieren wir zuerst mal *per Hand*, zumindest fangen wir mal damit an.

$$a(x) : g(x) =$$

$$(a_k x^k + a_{k-1} x^{k-1} + \ldots + a_1 x + a_0) : (x^L + g_{L-1} x^{L-1} + \ldots + g_1 x + g_0)$$
$$= a_k x^{k-L} + (a_{k-1} - a_k g_{L-1}) x^{k-1-L} + \ldots$$
$$\underline{a_k x^k + a_k g_{L-1} x^{k-1} + \ldots + a_k g_1 x^{k-L+1} + a_k g_0 x^{k-L}}$$
$$(a_{k-1} - a_k g_{L-1}) x^{k-1} + \ldots + (a_{k-L+1} - a_k g_1) x^{k-L+1} + (a_{k-L} - a_k g_0) x^{k-L} + a_{k-L-1} x^{k-L-1}$$
$$\underline{(a_{k-1} - a_k g_{L-1}) x^{k-1} + \ldots}$$

Wie üblich dividieren wir den höchsten Summanden in $a(x)$ – nämlich $a_k x^k$ – zuerst durch den höchsten Summanden in $g(x)$ – nämlich x^L – und multiplizieren das Ergebnis – nämlich $a_k x^{k-L}$ – wieder rückwärts mit $g(x)$. Das Resultat ziehen wir von $a(x)$ ab, sodass sich der höchste Summand in $a(x)$ – nämlich $a_k x^k$ – weghebt. Es ergibt sich also ein Polynom kleineren Grades, welches wir wieder durch x^L dividieren. So machen wir weiter, bis wir ein Polynom vom Grad $< L = \text{grad}(g(x))$ bekommen, welches wir nicht weiter dividieren können. Dies ist dann entweder 0, d. h. die Division geht auf, oder es ist eben der Rest bei Division von $a(x)$ durch $g(x)$. In Abb. 6.4 ist dieses Verfahren der Polynomdivision mittels Schieberegister umgesetzt.

Konkret funktioniert das wie folgt:

- Am Anfang sind alle Zustände (Zellen) mit 0 initialisiert.
- Danach werden der Reihenfolge nach $a_k, a_{k-1}, \ldots, a_0$ bei jedem Takt von links nach rechts in die Zellen geschoben.

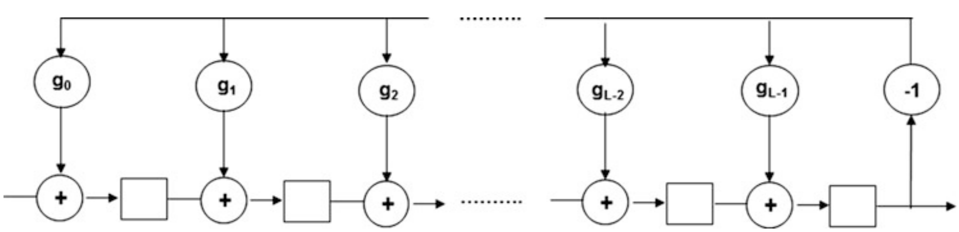


Abb. 6.4 Schieberegister zur Polynomdivision

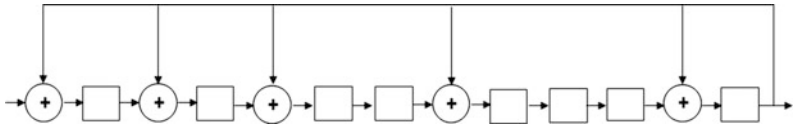


Abb. 6.5 Polynomdivision durch $g(x) = x^8 + x^7 + x^4 + x^2 + x + 1$

- Wegen der Initialisierung mit 0 wird beim Einschub von $a_k, a_{k-1}, \ldots, a_{k-L+1}$ stets 0 ausgegeben.
- Beim nächsten Takt des Schieberegisters wird dann ganz rechts sowohl a_k ausgegeben als auch $-a_k$ gebildet.
- $-a_k$ wird dann jeweils mit den g_j multipliziert und auf die Inhalte der Zellen links unterhalb der g_j-Skalierer addiert – also auf a_{k-L+j} –, wobei sich dies am Eingang auf den nächsten Inputkoeffizienten a_{k-L} bezieht.
- Nach dem Takt stehen dann in den Zellen die Werte $a_{k-L+j} - a_k g_j$. Das sind aber genau die Koeffizienten im Polynom unserer obigen Rechnung, mit denen wir die zweite Division durch x^L durchführen müssen. Der Wert a_{k-L-1} wartet nunmehr als Input für den nächsten Takt.
- Auf diese Art takten wir weiter. Nach $k - L + 1$ Schritten ist der Quotient rechts vollständig ausgegeben und der Rest der Division steht in den Zellen. Steht hier überall 0, so ist die Division aufgegangen.

6.2.5 Beispiel: Polynomdivision mittels Schieberegister

Wir erläutern das Vorgehen bei der Polynomdivision mittels Schieberegister wieder am Beispiel des Polynoms $g(x) = x^8 + x^7 + x^4 + x^2 + x + 1 \in \mathbb{Z}_2[x]$. Abb. 6.5 zeigt das zugehörige Schieberegister.

Wir wollen das Polynom $a(x) = x^{12} + x^{10} + x^5 + 1$ durch $g(x)$ dividieren. Wir überspringen in der unten stehenden Tabelle die ersten Takte, da hierbei nur die ersten acht Werte von $a(x)$ in das Register geschoben und gleichzeitig 0 herausgeschoben werden. Hier sind für die restlichen Takte die Input-, Zellen- und Outputwerte.

Takt	Input	Zellen	Output
Init	1 0 0 0 0 1 0 0 0 0 1 0 1	0 0 0 0 0 0 0 0	
⋮	⋮	⋮	
8	1 0 0 0 0	1 0 0 0 0 1 0 1	
9	1 0 0 0	1 0 1 0 1 0 1 1	1
10	1 0 0	1 0 1 1 1 1 0 0	1 1
11	1 0	0 1 0 1 1 1 1 0	0 1 1
12	1	0 0 1 0 1 1 1 1	0 0 1 1
13		0 1 1 1 1 1 1 0	1 0 0 1 1

Damit ergibt die Division von $a(x)$ durch $g(x)$ das Polynom $x^4 + x^3 + 1$ (im Output) mit dem Restpolynom $x^6 + x^5 + x^4 + x^3 + x^2 + x$ (in den Zellen).

6.2.6 Schieberegister über größeren Körpern

In unseren Beispielen haben wir nur binäre Polynome betrachtet und mussten daher in den Schieberegistern auch nur binär in $\mathbb{Z}_2$ rechnen. Was macht man aber bei größeren Körpern K, insbesondere bei $|K| = 2^m$? Wir erinnern uns an Abschn. 5.1, dass nämlich diese Körper als Reste modulo irreduzibler Polynome $f(x)$ in $\mathbb{Z}_2[x]$ konstruiert wurden, nämlich

$$K = \{g(x) \pmod{f(x)} | g(x) \in \mathbb{Z}_2[x] \text{ und } \operatorname{grad}(g(x)) < m\}.$$

Damit läuft also die Multiplikation zweier Körperelemente $g(x) \pmod{f(x)}$ und $h(x) \pmod{f(x)}$ in K auf die Polynommultiplikation $g(x)\, h(x)$ mit anschließender Polynomdivision durch $f(x)$ mit Rest hinaus, wobei dieser Rest $r(x)$ das gesuchte Produkt $r(x) = g(x)h(x) \pmod{f(x)}$ ist. Wenn man also mit Polynomen über dem Körper K mit 2^m Elementen arbeitet, so muss man einerseits die Körpermultiplikation und -division in K auf die eben beschriebene Weise mit Polynomen in $\mathbb{Z}_2[x]$ implementieren. Auf Basis dieses *Unterprogramms* kann man dann wiederum Polynome in $K[x]$ multiplizieren und dividieren.

Es ging in diesem Abschnitt nicht darum, eine komplette Theorie der Schieberegister zu erarbeiten, sondern vielmehr darum zu demonstrieren, wie effizient man mit ihnen gerade bei der Polynommultiplikation und -division rechnen kann. Die Algorithmen, die darauf beruhen – so auch insbesondere der euklidische Algorithmus, der durch wiederholte Division mit Rest den größten gemeinsamen Teiler bestimmt und dessen Vielfachsummendarstellung ausrechnet –, sind höchst effizient, schnell und leicht zu implementieren. Dies ist einer der wichtigsten Gründe, warum man sich besonders für zyklische Codes interessiert.

6.3 Codierung zyklischer Codes

Wir kommen nun zurück zu zyklischen Codes, speziell zu der Frage, wie man diese schnell und effizient codieren kann. Der Schlüssel hierzu ist natürlich das Generatorpolynom.

6.3.1 Codierung mit Generatorpolynom

Sei also C ein zyklischer $[n, k]$-Code mit Generatorpolynom $g(x)$. Wir wissen aus Abschn. 6.1, dass $\{g(x)x^{k-1}, \ldots, g(x)x, g(x)\}$ eine Basis von C ist. Wenn wir also ein

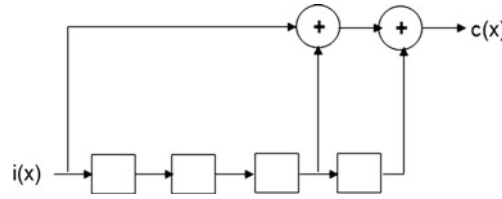


Abb. 6.6 Codierung des zyklischen Codes $Ham_2(4)$ mit Generatorpolynom $g(x) = x^4 + x + 1$

Informationstupel $i = (i_{k-1}, \ldots, i_0)$ in ein $c(x) \in C$ codieren wollen, so müssen wir wie üblich bei linearen Codes die Linearkombinationen der Basisvektoren bilden, also $c(x) = i_{k-1}g(x)x^{k-1} + \ldots + i_1 g(x)x + i_0 g(x) = (i_{k-1}x^{k-1} + \ldots + i_1 x + i_0)g(x)$ berechnen. Für das aus den Informationstupeln gebildete Polynom $i(x) = i_{k-1}x^{k-1} + \ldots + i_1 x + i_0$ bedeutet das, dass wir mit dem Generatorpolynom $g(x)$ multiplizieren müssen. Und das macht man – wie wir mittlerweile wissen – am besten mit Schieberegistern.

6.3.2 Beispiel: Codierung zyklischer Codes

Hamming-Code $Ham_2(4)$
Gleich wieder ein Beispiel zur Übung, und zwar ein solches, welches wir so noch nicht konstruiert haben. Man rechnet in $\mathbb{Z}_2[x]$ nach, dass $x^{15} + 1 = (x^4 + x + 1)(x^{11} + x^8 + x^7 + x^5 + x^3 + x^2 + x + 1)$ gilt. Das Generatorpolynom $g(x) = x^4 + x + 1 \in \mathbb{Z}_2[x]$ erzeugt uns also einen zyklischen Code der Länge 15, der Dimension 11 und mit Kontrollpolynom $h(x) = x^{11} + x^8 + x^7 + x^5 + x^3 + x^2 + x + 1$. Hier ist die Kontrollmatrix.

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

Die Spalten durchlaufen alle binären 4-Tupel ungleich $(0, 0, 0, 0)$, also ist unser zyklischer Code der Hamming-Code $Ham_2(4)$. Zum Codieren von $Ham_2(4)$ benutzen wir also das Schieberegister für die Polynommultiplikation mit $g(x)$. Abb. 6.6 zeigt das zugehörige Schaltbild.

Wir wählen wieder ein konkretes Rechenbeispiel. Dazu sei $i = (0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0)$ ein Informations-11-Tupel, also in Polynomschreibweise $i(x) = x^9 + x^8 + x^5 + x^3$. Wir *schieben* also jetzt unser Informationstupel taktweise in das Register und berechnen

so auch taktweise den Output. Die Zellen sind wieder mit 0 initialisiert.

Takt	Input	Zellen	Output
Init	0 0 0 1 0 1 0 0 1 1 0	0 0 0 0	
1	0 0 0 1 0 1 0 0 1 1	0 0 0 0	0
2	0 0 0 1 0 1 0 0 1	1 0 0 0	1 0
3	0 0 0 1 0 1 0 0	1 1 0 0	1 1 0
4	0 0 0 1 0 1 0	0 1 1 0	0 1 1 0
5	0 0 0 1 0 1	0 0 1 1	1 0 1 1 0
6	0 0 0 1 0	1 0 0 1	1 1 0 1 1 0
7	0 0 0 1	0 1 0 0	1 1 1 0 1 1 0
8	0 0 0	1 0 1 0	1 1 1 1 0 1 1 0
9	0 0	0 1 0 1	1 1 1 1 1 0 1 1 0
10	0	0 0 1 0	1 1 1 1 1 1 0 1 1 0
11		0 0 0 1	1 1 1 1 1 1 1 0 1 1 0
12	1. 0-Input	0 0 0 0	1 1 1 1 1 1 1 1 0 1 1 0
13	2. 0-Input	0 0 0 0	0 1 1 1 1 1 1 1 1 0 1 1 0
14	3. 0-Input	0 0 0 0	0 0 1 1 1 1 1 1 1 1 0 1 1 0
15	4. 0-Input	0 0 0 0	0 0 0 1 1 1 1 1 1 1 1 0 1 1 0

Wie nicht anders zu erwarten, wurde aus dem 11-Tupel $i = (0,1,1,0,0,1,0,1,0,0,0)$ ein 15-Tupel $c = (0,1,1,0,1,1,1,1,1,1,1,1,0,0,0)$. In Polynomschreibweise wurde also aus $i(x) = x^9 + x^8 + x^5 + x^3$ das Codewort $c(x) = x^{13} + x^{12} + x^{10} + x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3$.

6.3.3 Systematische Codierung zyklischer Codes

Etwas unschön an dem eben vorgestellten Verfahren ist allerdings, dass das Codewort $c(x)$ nicht in systematischer Form erscheint, d. h. an den ersten 11 Stellen die Informationsbits i_j und an den übrigen vier Stellen die Paritätsbits p_j stehen, die sich wiederum als Linearkombination aus den i_j berechnen. Der Grund hierfür ist klar. Die Generatormatrix, wie wir sie in Abschn. 6.1 aus dem Generatorpolynom abgeleitet haben, ist nicht in systematischer Form. Es wäre daher wünschenswert, auch ein Codierverfahren mittels Schieberegister zu haben, bei dem die Codewörter in systematischer Form erzeugt werden.

Sei wieder C ein zyklischer $[n, k]$-Code mit Generatorpolynom $g(x)$. Um sich die $c(x) \in C$ in systematischer Form zu beschaffen, muss es also gelingen, das Informationstupel $i = (i_{k-1}, \ldots, i_0)$ an die großen Potenzen $x^{n-1}, \ldots, x^{n-k}$ in $c(x)$ zu *schieben* und die zu berechnenden Paritätswerte an die kleineren Potenzen. Formal heißt das, man hätte zum Polynom $i(x) = i_{k-1}x^{k-1} + \ldots + i_1 x + i_0$ gern eine Darstellung

$$c(x) = x^{n-k} i(x) - t(x) \quad \text{und} \quad \text{grad}(t(x)) < n - k,$$

bei dem es $t(x)$ noch zu bestimmen gilt. Da im Codewort $c(x)$ ja der Faktor $g(x)$ enthalten ist, muss das gesuchte $t(x)$ die Bedingung $t(x) = x^{n-k} i(x) \,(\text{mod}\, g(x))$ erfüllen. Also können wir $t(x)$ als Rest der Division von $x^{n-k} i(x)$ durch $g(x)$ bestimmen.

Aber auch hierfür kennen wir ja bereits eine Schieberegisterschaltung. Allerdings soll unser Schieberegister nicht nur den Rest $t(x)$ bestimmen, sondern natürlich gleich auch das ganze Polynom $c(x)$ ausgeben, beginnend mit den Informationszeichen.

6.3.4 Beispiel: Systematische Codierung zyklischer Codes

Hamming-Code $Ham_2(4)$

Wir machen uns das Verfahren wieder am Beispiel $Ham_2(4)$ mit Generatorpolynom $g(x) = x^4 + x + 1 \in \mathbb{Z}_2[x]$ klar. In Abb. 6.7 beschreibt die obere Kette genau die

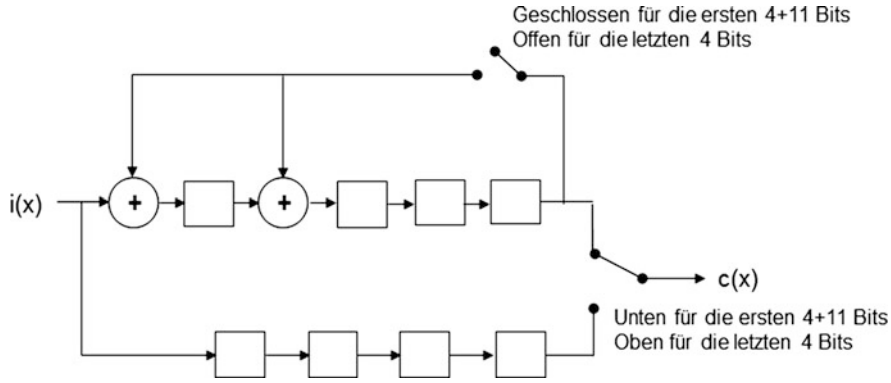


Abb. 6.7 Systematische Codierung des zyklischen Codes $Ham_2(4)$ mit Generatorpolynom $g(x) = x^4 + x + 1$

Division durch $g(x)$ mit Rest, allerdings ist ein zusätzlicher Brückenschalter eingebaut. Die untere Kette enthält keine Operationen, hier werden die Daten nur taktweise weitergeschoben. Und so funktioniert es genau.

- Alle Zellen sind mit 0 initialisiert.
- Das 11-Tupel $i(x)$ wird um 4 Stück 0-Bits verlängert, was der Multiplikation mit $x^{n-k} = x^4$ entspricht, und wartet so als Input.
- Der Brückenschalter ist geschlossen, der Kippschalter rechts ist nach unten gestellt.
- In den ersten vier Takten werden die ersten vier Werte von $i(x)x^4$ in das Register sowohl oben als auch unten geschoben. Im oberen Teil findet wegen der Initialisierung mit 0 keine verändernde Operation statt, im unteren Teil werden vier Nullen ausgegeben.
- Bei den nächsten elf Takten werden die übrigen Werte von $i(x)x^4$ in die Register oben und unten geschoben.
- Bei diesen elf Takten wird in der oberen Kette die Division von $i(x)x^4$ durch $g(x)$ mit Rest durchgeführt. Es wird dabei zwar nichts ausgegeben (wegen der Schalterstellung rechts), aber am Ende steht – wie wir wissen – genau der Rest der Division in den vier Zellen.
- Bei genau diesen elf Takten werden hingegen in der unteren Kette die elf ersten Werte von $i(x)x^4$ unverändert ausgegeben. Das sind aber genau die Werte von $i(x)$ selbst.
- Danach wird der obige Brückenschalter geöffnet und der rechte Kippschalter nach oben gestellt.
- In den letzten vier Takten wird dann schließlich der Inhalt der obigen Zellen ausgegeben. Das ist genau der Rest der Division von $i(x)x^4$ durch $g(x)$.
- Nach insgesamt 19 Takten steht im Output das Codewort $c(x)$ in systematischer Form. (Übrigens: Mit einer leicht modifizierten Schaltung geht das auch in nur 15 Takten.)

Wir wollen jetzt natürlich als Beispiel unser Informations-11-Tupel $i = (0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0)$ systematisch codieren. Wegen $i(x) = x^9 + x^8 + x^5 + x^3$ ist $i(x)x^4 = x^{13} + x^{12} + x^9 + x^7$. Um die Tabelle übersichtlicher zu gestalten, beschreiben wir nur die 15 Takte für die Division von $i(x)x^4$ durch $g(x)$ mit Rest. Denn wir wissen ja, dass das Schieberegister nach den vier führenden Nullen als Codewort c zunächst genau das 11-Tupel i ausgibt, gefolgt von dem 4-Tupel des zu berechnenden Rests. Das Ganze ergibt

dann c in systematischer Form.

Takt	Input	Obere Zellen
Init	0000000101001 10	0000
1	000000010100 11	0000
2	00000001010 01	1000
3	0000000101 00	1100
4	000000010 10	0110
5	00000001 01	0011
6	0000000 10	0101
7	000000 01	1110
8	00000 00	1111
9	0000 00	1011
10	000 00	1001
11	00 00	1000
12	0 00	0100
13	00	0010
14	0	0001
15		1100

Wir erhalten also als systematische Form das Codewort $c = (0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 1)$ bzw. in Polynomschreibweise $c(x) = i(x)x^4 + t(x) = x^{13} + x^{12} + x^9 + x^7 + x + 1$.

6.4 Meggitt-Decodierung

Natürlich wollen wir die zyklischen Codes auch wieder effizient decodieren. Darum kümmern wir uns in diesem Abschnitt.

6.4.1 Syndrompolynom und Syndromdecodierung für zyklische Codes

Sei C ein zyklischer $[n, k, d]$-Code mit Generatorpolynom $g(x)$. Wir erläutern nun zuerst das einfachste Decodierverfahren für zyklische Codes, die sog. **Syndromdecodierung**. Während die Syndromdecodierung bei linearem Code bekanntlich mit einer Kontrollmatrix arbeitet, verwendet man bei zyklischen Codes das Konzept des sog. Syndrompolynoms.

Für ein Wort $v(x) = v_{n-1}x^{n-1} + \ldots + v_0$ definiert man das **Syndrompolynom** $s(x) = s_v(x)$ als Rest von $v(x)$ bei Division durch $g(x)$, also $s_v(x) = v(x) \pmod{g(x)}$.

Dies lässt sich natürlich wieder effizient mittels Schieberegister berechnen. Ist $v(x) = c(x) + f(x)$ mit einem Codewort $c(x)$ und einem Fehlerpolynom $f(x)$, dann gilt also $s_v(x) = s_f(x)$, da $g(x)$ als Faktor in $c(x)$ enthalten ist. Das Syndrompolynom von $v(x)$ hängt daher nur vom Fehlerpolynom $f(x)$, nicht aber vom Codewort $c(x)$ ab.

Hier ist nun die entscheidende Tatsache, warum das Syndrompolynom hilfreich bei der Decodierung ist. Für die Fehlerkorrekturkapazität e von C (also $2e + 1 \leq d$) ist jedes Fehlerpolynom vom Gewicht $wt(f(x)) \leq e$ durch sein Syndrompolynom $s_f(x)$ eindeutig bestimmt.

Das ist leicht einzusehen. Nehmen wir dazu zwei Fehlerpolynome $f(x)$ und $f'(x)$ mit demselben Syndrompolynom $s(x)$. Dann gilt $f(x) = s(x) = f'(x) \pmod{g(x)}$, also $f(x) - f'(x) \in C$, da $g(x)$ als Faktor in $f(x) - f'(x)$ enthalten ist. Wegen $wt(f(x)) \leq e$ und $wt(f'(x)) \leq e$ gilt jedoch $wt(f(x) - f'(x)) < d$. Aber jedes von 0 verschiedene Codewort in C hat ja Gewicht mindestens d. Also ist $f(x) = f'(x)$.

Und so funktioniert das zugehörige Decodierverfahren. Man stellt eine Liste aller Polynome vom Gewicht $\leq e$ auf samt deren Syndrompolynome, die sog. **Syndromauswertetabelle**. Diese Tabelle enthält also alle potenziellen Fehlerpolynome. Für binäre Codes könnte sie etwa so beginnen:

$f(x)$	1	x	x^2	...	$1+x$	$1+x^2$	...
$s(x)$	$1 \pmod{g(x)}$	$x \pmod{g(x)}$	$x^2 \pmod{g(x)}$	...	$1+x \pmod{g(x)}$	$1+x^2 \pmod{g(x)}$	...

Die Tabelle kann man ein für alle Mal für C erstellen und abspeichern. Dabei haben verschiedene Fehlervektoren auch verschiedene Syndrompolynome. Operativ geht man so vor. Für ein ankommendes Wort $v(x) = c(x) + f(x)$ berechnet man das Syndrompolynom $s_v(x)$, das ja dem Syndrompolynom $s_f(x)$ des zu bestimmenden Fehlerpolynoms $f(x)$ entspricht. Dann sucht man in der Syndromauswertetabelle das eindeutige Fehlerpolynom $f(x)$ mit diesem Syndrompolynom. Findet man ein solches, so korrigiert man $v(x)$ zu $c(x) = v(x) - f(x)$. Findet man kein Fehlerpolynom in der Tabelle, so weiß man, dass mehr als e Fehler aufgetreten sind, und man korrigiert nicht. Dieses Verfahren ist also wieder ein Beispiel einer BD-Decodierung (Bounded Distance). Man kann sich leicht vorstellen, dass die Syndromauswertetabelle und die damit verbundene Durchsuchung sehr umfangreich sein können und man gerade hier ansetzt, schnellere Verfahren zu entwickeln.

6.4.2 Kanaldecodierung – Die Kanalcodierung rückgängig machen

Nachdem man – etwa mittels Syndromdecodierung – aus dem empfangenen Wort $v(x)$ das entsprechende Codewort $c(x) \in C$ bestimmt hat, will man aber auch wieder die ursprüngliche Information $i(x)$ ermitteln, d. h. die Codierung rückgängig machen. Im Falle dass C nicht in systematischer Form erzeugt wurde, d. h. mit dem Generatorpolynom $g(x)$ multipliziert wurde, muss man also das Codewort $c(x)$ wieder durch $g(x)$ teilen.

Das macht man wieder am schnellsten mit dem Schieberegister, welches $c(x)$ durch $g(x)$ mit Rest dividiert, nur dass jetzt eben kein Rest bleibt. Ist C dagegen in systematischer Form codiert, so kann man $i(x)$ direkt an den k höchsten Koeffizienten von $c(x)$ ablesen.

6.4.3 Meggitt-Algorithmus und -Decodierung

Nun zu einer Optimierung der Syndromdecodierung. Die **Meggitt-Decodierung** setzt genau an deren Schwachstelle an, nämlich an der umfangreichen Syndromauswertetabelle, die durchsucht werden muss. Bei der Meggitt-Decodierung wird es ausreichen, nur einige typische Fehlerpolynome und deren Syndrompolynome in der Tabelle vorzuhalten. Das hierzu entscheidende Argument benutzt die algebraische Struktur zyklischer Codes.

Wir betrachten wieder einen zyklischen $[n, k]$-Code C mit Generatorpolynom $g(x)$ und beschreiben den **Meggitt-Algorithmus**. Sei dazu $s_v(x)$ das Syndrompolynom eines Wortes $v(x)$, also $s_v(x) = v(x) \,(\mathrm{mod}\,(g(x))$. Sei weiterhin $w(x) = xv(x) \,(\mathrm{mod}\,(x^n - 1))$ das Wort, das durch zyklische Vertauschung der Koeffizienten um eine Stelle aus $v(x)$ hervorgeht, und $s_w(x) = w(x) \,(\mathrm{mod}\, g(x))$ dessen Syndrompolynom. Dann ist $s_w(x) = xs_v(x) \,(\mathrm{mod}\,(g(x))$.

Ob wir also zuerst durch Multiplikation mit x die Koeffizienten von $v(x)$ zyklisch vertauschen und dann davon das Syndrompolynom bilden oder zuerst von $v(x)$ das Syndrompolynom berechnen, dieses dann mit x multiplizieren und hiervon wieder das Syndrompolynom bilden, kommt auf das Gleiche hinaus. Um diese Kernaussage des Meggitt-Algorithmus nachzuweisen, sammeln wir zuerst einmal, was wir alles wissen. Mit $v(x) = v_{n-1}x^{n-1} + \ldots + v_0$ gelten nämlich folgende Gleichungen für geeignete Polynome $q_1(x)$ und $q_2(x)$:

- $v(x) = g(x)q_1(x) + s_v(x)$, folglich $xv(x) = xg(x)q_1(x) + xs_v(x)$,
- $w(x) = xv(x) - (x^n - 1)v_{n-1} = xv(x) - h(x)g(x)v_{n-1}$ mit dem Kontrollpolynom $h(x)$ von C,
- $xs_v(x) = g(x)q_2(x) + t(x)$, wobei $t(x)$ mit $\mathrm{grad}(t(x)) < \mathrm{grad}(g(x))$ das Syndrompolynom von $xs_v(x)$ ist.

Zusammen ergibt sich $w(x) = (xq_1(x) + q_2(x) - v_{n-1}h(x))g(x) + t(x) = q_3(x)g(x) + t(x)$ mit einem Polynom $q_3(x)$. Dies bedeutet aber, dass $t(x)$ auch das Syndrompolynom $s_w(x)$ von $w(x)$ ist.

Wir kommen nun zur **Meggitt-Decodierung**. Vertauscht man bei einem Fehlerpolynom in der Syndromauswertetabelle dessen Positionen zyklisch, so haben alle solche wieder Gewicht $\leq e$ und kommen daher ebenfalls in der Syndromauswertetabelle vor. Bei der **Meggitt-Tabelle** wird hingegen zu jedem solchen Zyklus nur ein Vertreter in die

Tabelle aufgenommen, und zwar ein solcher mit höchstem Koeffizienten ungleich 0 (bei der Potenz x^{n-1}). Die Meggitt-Tabelle kann ebenfalls ein für alle Mal für C aufgestellt und abgespeichert werden.

Operativ kann man sich das Ganze nun so vorstellen. Für ein ankommendes Wort $v(x) = c(x) + f(x)$ berechnet man das Syndrompolynom $s(x)$, das ja dem Syndrompolynom des zu bestimmenden Fehlerpolynoms $f(x)$ entspricht. Dann wählt man in der Meggitt-Tabelle ein Fehlerpolynom $f^\circ(x)$ – am besten zuerst eines von kleinstem Gewicht – und überprüft, ob dessen Syndrompolynom $s^\circ(x)$ mit dem berechneten $s(x)$ übereinstimmt. Falls ja, dann korrigiert man mit $f(x) = f^\circ(x)$. Falls nicht, so berechnet man das Syndrompolynom von $xs(x)$. Wir wissen auf Basis des Meggitt-Algorithmus, dass dies auch das Syndrompolynom von $xf(x) \pmod{x^n - 1}$ ist, also des Polynoms mit zyklischer Vertauschung der Koeffizienten von $f(x)$ um eine Stelle. Ist also das Syndrompolynom von $xs(x)$ gleich $s^\circ(x)$, so müssen auch die zugehörigen Fehlerpolynome gleich sein, also $f^\circ(x) = xf(x) \pmod{x^n - 1}$. Wir folgern schließlich, dass das gesuchte Fehlerpolynom $f(x)$ durch zyklische Vertauschung der Koeffizienten um eine Stelle aus $f^\circ(x)$ hervorgeht, wobei dieser Zyklus in die entgegengesetzte Richtung wie oben zu verstehen ist. Auf diese Weise operiert man schrittweise weiter durch fortgesetzte Multiplikation mit x, bis man den gesamten Vertauschungszyklus der Koeffizienten in $f(x)$ durchlaufen hat. Hier sind tabellarisch zusammengefasst die iterativen Syndrompolynomberechnungen und zugeordneten Fehlerpolynome.

Syndrompolynom	Falls $s_i(x) = s^\circ(x)$, so $f^\circ(x) =$
$s_0(x) = s(x) =$ Syndrompolynom von $v(x)$	$f(x) = f_{n-1}x^{n-1} + \ldots + f_0$
$s_1(x) =$ Syndrompolynom von $xs_0(x)$	$xf(x) \pmod{x^n - 1} = f_{n-2}x^{n-1} + \ldots + f_0x + f_{n-1}$
$s_2(x) =$ Syndrompolynom von $xs_1(x)$	$x^2 f(x) \pmod{x^n - 1} = f_{n-3}x^{n-1} + \ldots + f_{n-1}x + f_{n-2}$
$\vdots$	$\vdots$
$s_i(x) =$ Syndrompolynom von $xs_{i-1}(x)$	$x^i f(x) \pmod{x^n - 1}$
$\vdots$	$\vdots$
$s_{n-1}(x) =$ Syndrompolynom von $xs_{n-2}(x)$	$x^{n-1} f(x) \pmod{x^n - 1} = f_0x^{n-1} + \ldots + f_2x + f_1$

Ist also das iterierte Syndrompolynom $s_i(x)$ gleich $s^\circ(x)$, so ist $f^\circ(x) = x^i f(x) \pmod{x^n - 1}$ und $f(x)$ ist das i-fach zyklisch rücklaufend verschobene Polynom $f^\circ(x)$. Findet man dabei nicht das gesuchte Fehlerpolynom $f(x)$, so startet man das Verfahren neu mit einem weiteren Fehlerpolynom in der Meggitt-Tabelle – wieder mit möglichst kleinem Gewicht – und dessen n-fach-Zyklus.

In der Praxis implementiert man das Verfahren allerdings in einer anderen Reihenfolge. Man vergleicht im ersten Takt das Syndrompolynom $s(x)$ mit den Syndrompolynomen der gesamten Meggitt-Tabelle. Hat man hierbei das Fehlerpolynom noch nicht gefunden, berechnet man sukzessive in weiteren Takten die Syndrompolynome von $s_i(x)$ und vergleicht diese wieder mit der gesamten Meggitt-Tabelle, und zwar solange, bis man eine Übereinstimmung und damit den gesuchten Fehlervektor findet. Dieses Verfahren

lässt sich auch leicht mit einer Schieberegisterschaltung realisieren, die zunächst $v(x)$ und dann iterativ $xs_i(x)$ durch $g(x)$ mit Rest teilt und dann bei jedem Takt die Suche durchführt. Jedenfalls überprüft man auf diese Art implizit *alle* Fehlerpolynome der Syndromauswertetabelle, ohne jedoch stets die gesamte Liste der potenziellen Fehlerpolynome durchsuchen zu müssen. Findet man auf diese Weise insgesamt kein Fehlerpolynom, so weiß man wieder, dass mehr als e Fehler aufgetreten sind, und man korrigiert nicht.

In der Tat geht man allerdings meist so vor, dass man bei jedem Takt der Reihe nach immer nur einen weiteren Koeffizienten des gesuchten Fehlerpolynoms bestimmt. Hat man nämlich beim j. Takt eine Übereinstimmung der Syndrompolynome gefunden, so gibt man den höchsten Koeffizienten des in der Meggitt-Tabelle zugehörenden Polynoms aus (nämlich den bei x^{n-1}), sonst aber 0. Dieser ist dann nämlich der j-höchste Koeffizient des gesuchten Fehlerpolynoms $f(x)$. Dies erscheint auf den ersten Blick kurios, da man bei der ersten gefundenen Übereinstimmung eigentlich bereits das gesamte Fehlerpolynom identifiziert hat. Trotzdem ist diese Schaltung effektiver.

6.4.4 Beispiel: Meggitt-Decodierung

Hamming-Code $Ham_2(4)$
Wir betrachten als Beispiel wieder $Ham_2(4)$ von Länge 15, Dimension 11 und mit Generatorpolynom $g(x) = x^4 + x + 1$. Wegen seines Minimalabstands $d = 3$ kann der Code nur einen Fehler korrigieren und daher dürfen die potenziellen Fehlerpolynome in der Syndromauswertetabelle und der Meggitt-Tabelle nur einen Summanden haben. In der Syndromauswertetabelle würden also alle Polynome $x^{14}, \ldots, x^2, x, 1$ aufzulisten sein, in der Meggitt-Tabelle genügt das einzige Fehlerpolynom $f^\circ(x) = x^{14}$. Man rechnet mittels Division mit Rest durch $g(x)$ nach, dass das Syndrompolynom $s^\circ(x)$ gleich $x^3 + 1$ ist.

Wir senden nun ein Codewort aus $Ham_2(4)$, etwa das im letzten Abschnitt bereits codierte Polynom $c(x) = x^{13} + x^{12} + x^{10} + x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3$, und nehmen an, dass bei der Übertragung ein Fehler bei x^{13} aufgetreten ist, also ist $v(x) = c(x) + f(x) = x^{12} + x^{10} + x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3$ das empfangene Wort. Nun beginnt die Meggitt-Decodierung. Dazu berechnen wir zunächst das Syndrompolynom von $v(x)$, d. h. wir dividieren $v(x)$ mit Rest durch $g(x)$, und erhalten $s(x) = x^3 + x^2 + 1$. Dies ist leider nicht gleich $s^\circ(x)$. Deshalb berechnen wir das Syndrompolynom von $xs(x) = x^4 + x^3 + x$. Dies ist $x^3 + 1$, also gleich $s^\circ(x)$ und wir folgern daher $x^{14} = f^\circ(x) = xf(x) \pmod{x^{15} + 1}$, d. h. $f(x) = x^{13}$. Genau dort hatten wir ja auch den Fehler eingebaut.

6.4.5 Komplexität: Meggitt-Decodierung

Die Meggitt-Decodierung nutzt also die Tatsache aus, dass die Suche in kleinen Listen zusammen mit einfachen und effizienten Rechenalgorithmen allemal schneller ist als die

Suche in – und der damit verbundene Zugriff auf – umfangreichen Listen. Hier auch noch eine kurze Bemerkung zur Komplexität der Meggitt-Decodierung. Sei dazu C ein zyklischer $[n,k,d]_q$-Code mit Generatorpolynom $g(x)$, also $\text{grad}(g(x)) = n-k$. Weiter sei C e-fehlerkorrigierend, also $2e+1 \leq d$. Wir schauen uns wieder die Anzahl der elementaren Operationen bei der Meggitt-Decodierung an. Die Berechnung von $s(x)$ benötigt $2(n-k)k$ und die iterierte Berechnung der Syndrompolynome mittels Division durch $g(x)$ mit Rest weitere $2(n-k)n$ elementare Operationen, also insgesamt $\sim 2(n-k)(n+k)$ elementare Operationen. Die Anzahl m der potenziellen Fehlerpolynome in der gesamten Syndromauswertetabelle ist jedenfalls $m = n(q-1) + \binom{n}{2}(q-1)^2 + \ldots + \binom{n}{e}(q-1)^e$, in der Meggitt-Tabelle also nur m/n. Zählt man nun noch die notwendigen Vergleiche für die n iterierten Syndrompolynome, so ergeben sich also insgesamt $\sim n(n-k)m/n + 2(n-k)(n+k)$ elementare Operationen. Für *kleine e* gegenüber n bedeutet dies eine Komplexität von $\sim (n-k)n^e(q-1)^e + 2(n-k)(n+k)$. Betrachten wir den üblichen Anwendungsfall $q = 2$ und nehmen $k \sim n/2$, so ist die asymptotische Komplexität $\sim n^{e+1}/2 + (3/2)n^2$.

Da zyklische Codes ja auch lineare Codes sind, hätte man auch die altbekannte Syndromdecodierung für lineare Codes (mit Kontrollmatrix, nicht mit Syndrompolynom) verwenden können. Diese hat asymptotisch die Komplexität $\sim (n-k)q^{n-k}$, im binären Fall und bei $k \sim n/2$ also $\sim n2^{n/2-1}$ und ist somit exponentiell in n.

6.5 CRC – Cyclic Redundancy Check

6.5.1 ARQ-Verfahren

Bislang haben wir uns im Wesentlichen mit der Frage beschäftigt, wie man auftretende Fehler oder Auslöschungen bei der Datenübertragung oder beim Lesen von Datenträgern automatisch korrigieren kann. Dies nennt man wie schon erwähnt **FEC-Verfahren** (**Forward Error Correction**). Es gibt aber auch Anwendungen, bei denen lediglich die Fehlererkennung im Vordergrund steht. **ARQ-Verfahren** (**Automatic Repeat reQuest**, dt. automatische Wiederholungsanfrage) werden insbesondere bei Computernetzen eingesetzt, um eine zuverlässige Datenübertragung durch Sendewiederholungen zu gewährleisten. Durch die Möglichkeit der Fehlererkennung kann ein Empfänger aufgetretene Übertragungsfehler feststellen, die er dann dem Sender über einen Rückkanal mitteilen muss. Man verwendet hierzu sog. **ACK/NAK**-Signale (**Acknowledgement** bzw. **Negative Acknowledgement**, d. h. korrekter Empfang bestätigt bzw. Wiederholungsanfrage). Im Falle eines NAK-Signals wird die Nachricht solange wiederholt, bis der Empfänger keinen Fehler mehr erkennt. Beim **Stop-and-wait**-Verfahren wartet der Sender nach jedem Datenpaket auf das ACK/NAK-Signal, bevor er weiter sendet. Falls er keine Reaktion über den Rückkanal innerhalb eines bestimmten Zeitrahmens erhält (sog. Time-out), sendet er das Datenpaket automatisch neu. Das **Go-back-*n***-Verfahren dagegen erlaubt es dem Sender, n Datenpakete zu senden, bevor die Bestätigung für die erste Einheit durch den Empfänger

erfolgt sein muss. Kommt es bei diesem Verfahren beim Warten auf die Bestätigungen zu einem Time-out, so übermittelt der Sender die Datenpakete des gesamten Zeitfensters neu.

6.5.2 FCS- und CRC-Verfahren

Um eine Informationseinheit auf Fehler überprüfen zu können, wird eine Folge von Prüfbits als **FCS (Frame Checking Sequence)** an die Informationssequenz angehängt. Die einfachste Form kennen wir schon, den Paritätsprüfungscode, bei dem nur ein Bit zur Paritätsprüfung angehängt wird und bei dem so *ein* Fehler erkannt werden kann.

Das FCS-Verfahren funktioniert bei mehreren Bits (oder Elementen aus einem größeren Körper K) am besten, wenn man das Prüfverfahren durch zyklische Codes realisiert. Es heißt dann **CRC (Cyclic Redundancy Check).** Der CRC wurde 1961 von **William Wesley Peterson** (1924–2009) entwickelt, einem amerikanischer Mathematiker und Informatiker, der uns auch später noch beim PGZ-Decodierverfahren begegnen wird. Wie aber funktioniert das Verfahren? Sei dazu C wieder ein zyklischer $[n, k]$-Code über K mit Generatorpolynom $g(x)$ und $m = \text{grad}(g(x)) = n - k$. Dann wird bei CRC ein Informationstupel $i = (i_{k-1}, \ldots, i_0)$, in Polynomdarstellung also $i(x) = i_{k-1}x^{k-1} + \ldots + i_0$, in systematischer Form codiert, das Codewort $c(x) \in C$ hat also die Gestalt $c(x) = x^m i(x) - t(x)$ mit $\text{grad}(t(x)) < m$. Dabei ist $t(x)$ der Rest der Division von $x^{n-k}i(x) = x^m i(x)$ durch $g(x)$ und $c(x)$ ist damit ein Vielfaches von $g(x)$, so wie wir das in Abschn. 6.3 besprochen haben. Der Empfänger kann also das Informationstupel $i = (i_{k-1}, \ldots, i_0)$ an den großen Potenzen $x^{n-1}, \ldots, x^{n-k}$ von $c(x)$ direkt ablesen und an den kleineren Potenzen stehen genau $n - k = m$ Paritätswerte. Bevor der Empfänger die Information an einem empfangenen Wort $v(x)$ aber abliest, muss er überprüfen, ob Fehler bei der Übertragung aufgetreten sind, ob also $v(x)$ überhaupt ein gültiges Codewort ist. Dies tut er, indem er $v(x)$ durch $g(x)$ dividiert. Geht die Division nicht auf, so ist $v(x)$ kein Codewort, d. h. es sind Fehler aufgetreten, und die automatische Wiederholungsanfrage (ARQ) wird gestartet. Dabei sind also sowohl die Codierung als auch die Überprüfung mit genau derselben Schieberegisterschaltung „Division durch $g(x)$" höchst effizient und schnell implementierbar.

6.5.3 CRC-Codes und -Polynome

Der Anwendung entsprechend werden CRC-Codes nur über $\mathbb{Z}_2$ definiert. Ein **CRC-Code** ist also ein

- binärer zyklischer Code mit einem **CRC-Polynom**,
- mithilfe dessen systematisch codiert wird

- und bei dem die Fehlererkennung mittels Division durch das CRC-Polynom erfolgt.

Die in der Praxis verwendeten CRC-Polynome sind von der Form $g(x) = (x+1)m(x)$ oder $g(x) = m(x)$ mit einem irreduziblen Polynom $m(x) \in \mathbb{Z}_2[x]$ vom Grad > 1.

Es gibt eine Reihe von – meist international genormten – CRC-Polynomen. Hier sind die wichtigsten, denen wir auch allen später noch in Praxisanwendungen begegnen werden.

CRC-5-USB	$g(x) = x^5 + x^2 + 1$ $n = 2^5 - 1 = 31$ Hamming-Code $Ham_2(5)$
CRC-5-Bluetooth	$g(x) = x^5 + x^4 + x^2 + 1 = (x+1)(x^4 + x + 1)$ $n = 2^4 - 1 = 15$ $x^4 + x + 1$ erzeugt $Ham_2(4)$
CRC-7	$g(x) = x^7 + x^3 + 1$ $n = 2^7 - 1 = 127$ Hamming-Code $Ham_2(7)$
CRC-8-CCITT	$g(x) = x^8 + x^2 + x + 1 = (x+1)(x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + 1)$ $n = 2^7 - 1 = 127$
CRC-8-SAE-J1850	$g(x) = x^8 + x^4 + x^3 + x^2 + 1$ $n = 2^8 - 1 = 255$
CRC-8-Bluetooth	$g(x) = x^8 + x^7 + x^5 + x^2 + x + 1 = (x+1)(x^7 + x^4 + x^3 + x^2 + 1)$ $n = 2^7 - 1 = 127$
CRC-16-IBM	$g(x) = x^{16} + x^{15} + x^2 + 1 = (x+1)(x^{15} + x + 1)$ $n = 2^{15} - 1 = 32.767$
CRC-16-CCITT	$g(x) = x^{16} + x^{12} + x^5 + 1$ $= (x+1)(x^{15} + x^{14} + x^{13} + x^{12} + x^4 + x^3 + x^2 + x + 1)$ $n = 2^{15} - 1 = 32.767$
CRC-24-Q	$g(x) = x^{24}+x^{23}+x^{18}+x^{17}+x^{14}+x^{11}+x^{10}+x^7+x^6+x^5+x^4+x^3+x+1$ $= (x+1)(x^{23} + x^{17} + x^{13} + x^{12} + x^{11} + x^9 + x^8 + x^7 + x^5 + x^3 + 1)$ $n = 2^{23} - 1$
CRC-32	$g(x) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$ $n = 2^{32} - 1$

6.5.4 Erkennung von Bündelfehlern

Man spricht von einem ***b*-Bündelfehler**, wenn in einem empfangenen Wort bis zu b Fehler auftreten, und zwar an maximal b aufeinanderfolgenden Positionen. Bündelfehler

treten z. B. bei Impulsstörungen im Kabelnetzwerk oder in Abschattungen bei drahtloser Funkübertragung auf. Wir hatten uns schon mal mit Fehlerbündeln beschäftigt. Der Begriff Fehlerbündel wird dabei eher unpräzise bei Häufungen von Fehlern benutzt, möglicherweise auch zwei oder mehreren Häufungen pro Wort oder auch noch zusätzlichen Einzelfehlern je Wort außerhalb der Häufungen und das Ganze womöglich noch über mehrere Wörter verteilt. Bei b-Bündelfehlern betrachtet man konkreter den gesamten Bereich vom ersten bis zum letzten in einem Wort auftretenden Fehler, der dann maximal die Länge b haben darf. Bei Fehlerbündeln hatten wir mit Interleaving die Fehler korrigiert. Bei ARQ zur Fehlererkennung benötigen wir aber das viel schnellere CRC-Verfahren. Hier ist die wichtigste Fehlererkennungseigenschaft aller CRC-Codes.

CRC-Codes mit CRC-Polynom $g(x)$ können b-Bündelfehler bis zur Länge $b \leq \text{grad}(g(x))$ erkennen.

Dies sieht man so. Sei $v(x) = c(x) + f(x)$ das empfangene Wort. Das Fehlerpolynom lässt sich dann schreiben als $f(x) = a_{i+b-1}x^{i+b-1} + a_{i+b-2}x^{i+b-2} + \ldots + a_i x^i = x^i(a_{i+b-1}x^{b-1} + \ldots + a_i)$ für ein i. Würde der b-Bündelfehler nicht erkannt werden, so wäre $g(x)$ ein Teiler von $v(x)$ und damit von $f(x)$. Da aber $g(x)$ jedenfalls kein Teiler von x^i ist, müsste $g(x)$ das Polynom $a_{i+b-1}x^{b-1} + \ldots + a_i$ teilen. Das geht aber nicht wegen $b - 1 < \text{grad}(g(x))$.

6.5.5 Zusätzliche Fehlererkennung bei CRC-16 und CRC-24

Die CRC-16- und CRC-24-Codes können

- jede ungerade Anzahl von Fehlern sowie auch
- zwei einzelne – möglicherweise weit auseinander liegende – Fehler erkennen.

Wir überlegen uns auch diese Aussage. Da $x + 1$ ein Teiler von $g(x)$ ist, folgt $c(1) = 0$ für jedes Codewort $c(x)$. Bei einer ungeraden Anzahl von Fehlern im empfangenen Wort $v(x) = c(x) + f(x)$ gilt jedoch $v(1) = f(1) = 1$.

Enthält das empfangene Wort andererseits genau zwei Fehler, also $f(x) = x^i + x^j$ mit $i > j$, so würde bei Nichterkennung der Fehler $g(x)$ ein Teiler von $x^j(x^{i-j} + 1)$ sein, also müsste $g(x)$ das Polynom $x^{i-j} + 1$ teilen. Da die beiden CRC-16-Polynome jeweils einen Code der Länge $n = 2^{15} - 1 = 32.767$ erzeugen, ist $g(x)$ jedenfalls ein Teiler von $x^{32767} + 1$. Man muss nun rechnerisch überprüfen, dass $g(x)$ kein Teiler von $x^m + 1$ für jedes $m < 32.767$ ist. Bei CRC-24 gilt entsprechendes für $x^m + 1$ mit $m < 2^{23} - 1$. Damit kann $g(x)$ auch kein Teiler von $x^{i-j} + 1$ sein.

6.5.6 Beispiel: CRC-Codierung

Das CRC-16-IBM-Verfahren
Wir wollen ein konkretes Rechenbeispiel zur Codierung mit CRC-16-IBM vorführen. Wegen $(x^n+1)^2 = (x^{2n}+1)$ kann man grundsätzlich Bitfolgen *beliebiger* Länge – z. B. einen Dateiinhalt – mit einer CRC-Prüfsequenz versehen, jedoch wird in der Praxis meist *blockweise* vorgegangen. Unsere Beispielnachricht sei daher kürzer, nämlich $i = 10101101$ und folglich $i(x) = x^7 + x^5 + x^3 + x^2 + 1$, wobei wir die Bitfolge mit absteigenden x-Potenzen identifiziert haben. Nun muss das Polynom $x^{16}i(x) = x^{23}+x^{21}+x^{19}+x^{18}+x^{16}$ durch $g(x) = x^{16} + x^{15} + x^2 + 1$ mit Rest dividiert werden.

$$
\begin{array}{l}
x^{23} + x^{21} + x^{19} + x^{18} + x^{16} : x^{16} + x^{15} + x^2 + 1 = x^7 + x^6 + x^3 + 1 \\
\underline{x^{23} + x^{22} + x^9 + x^7} \\
\qquad x^{22} + x^{21} + x^{19} + x^{18} + x^{16} + x^9 + x^7 \\
\qquad \underline{x^{22} + x^{21} + x^8 + x^6} \\
\qquad\qquad x^{19} + x^{18} + x^{16} + x^9 + x^8 + x^7 + x^6 \\
\qquad\qquad \underline{x^{19} + x^{18} + x^5 + x^3} \\
\qquad\qquad\qquad x^{16} + x^9 + x^8 + x^7 + x^6 + x^5 + x^3 \\
\qquad\qquad\qquad \underline{x^{16} + x^{15} + x^2 + 1} \\
\qquad\qquad\qquad\qquad x^{15} + x^9 + x^8 + x^7 + x^6 + x^5 + x^3 + x^2 + 1
\end{array}
$$

Also ist $x^{16}i(x) = x^{15} + x^9 + x^8 + x^7 + x^6 + x^5 + x^3 + x^2 + 1 (\mathrm{mod}\, g(x))$ und das gesuchte Codewort lautet $c(x) = x^{23}+x^{21}+x^{19}+x^{18}+x^{16+}x^{15}+x^9+x^8+x^7+x^6+x^5 + x^3 + x^2 + 1$. Dies liest sich als Bitfolge wie 10101101 – *Ende Nachricht, Beginn FCS* – 1000001111101101.

6.5.7 Anwendung: CRC als Codeschutz

Wie zu Beginn dieses Abschnitts bereits erwähnt, liegt die Hauptanwendung des CRC-Verfahrens in der Netzwerktechnologie. Aber auch bei Speicherchips und Schnittstellen findet es Anwendung, wie wir ausführlich im nächsten Abschnitt erläutern werden. Hier ein kurzer Ausblick auf eine andere Anwendung, die wir viel später erst sehen werden, die aber auf den ersten Blick etwas verblüffend erscheint. CRC wird auch als Schutz anderer Codes eingesetzt. Das geht nach folgender Methode. Man codiert eine Nachricht mit einem Code, in der Regel ist das dann ein Faltungscode oder Turbocode, die wir erst in Kap. 9 und 10 kennenlernen werden. Beim Decodieren kann es aber passieren, dass der ein oder andere Decodierfehler auftritt, den man als solchen gar nicht leicht bemerkt und den man auch nicht mit Methoden wie BD-Decodierung isolieren kann. Daher nutzt man einen CRC-Code als äußeren Code einer Codeverkettung, d. h. zuerst wird CRC auf die Nachricht oder auch nur auf die wichtigsten Teile davon angewendet, dann der Faltungs-

code bzw. Turbocode als innerer Code – ähnlich wie beim Interleaving. Nach Decodierung des Faltungs-/Turbocodes überprüft man das Ergebnis mit CRC. Liefert dieser Test *nicht o. k.*, so weiß man, dass es sich nicht um ein korrektes Codewort handeln kann. Dann ignoriert man dieses Wort einfach und hofft, dass dies nicht allzu häufig geschieht und sich daher nicht allzu störend auf die Gesamtnachricht auswirkt, oder man interpoliert den Wert – sofern dies Sinn macht – wie bei der Audio-CD oder auch bei Fotos.

6.6 Computernetzwerke, Schnittstellen und Speicherchips

Wir hatten im letzten Abschnitt das **ARQ-Verfahren** (Automatic Repeat reQuest) vorgestellt und wie es mit CRC-Polynomen umgesetzt wird. Jetzt wollen wir uns wieder um die praktische Anwendung kümmern und beginnen dazu mit Computernetzwerken und deren Netzwerkprotokollen.

6.6.1 OSI-Referenzmodell

Ein Netzwerkprotokoll besteht aus einem Satz von Regeln und Formaten (Syntax), die das Kommunikationsverhalten der kommunizierenden Instanzen in den Computern bestimmen. Zur Systematisierung hat sich hierzu das sog. **OSI-Referenzmodell** durchgesetzt, das die Protokolle in sieben Ebenen unterteilt.

Schicht 7 Anwendungsschicht
Schicht 6 Darstellungsschicht
Schicht 5 Sitzungsschicht
Schicht 4 Transportschicht
Schicht 3 Vermittlungsschicht
Schicht 2 Sicherungsschicht
Schicht 1 Bitübertragungsschicht

Es würde den Rahmen sprengen, dies hier zu vertiefen, nur so viel: Je rudimentärer eine Kommunikation ist, desto weniger Ebenen sind notwendig bzw. umso rudimentärer können die Protokolle auf den oberen Ebenen ausfallen. Läuft z. B. eine Kommunikation völlig ohne Anwender ab, so ist keine Darstellungs- und Anwendungsschicht notwendig. Handelt es sich um feste Punkt-zu-Punkt-Verbindungen, so benötigt man keine Vermittlungs- und Transportschicht.

6.6.2 Anwendung: Die beiden CRC-16-Verfahren

Die wohl bekanntesten ARQ-Verfahren, die auf den beiden CRC-16-Polynomen beruhen, erweitern ein Informationspaket um 16 Paritätsbits zur Fehlererkennung.

CRC-16-IBM war ursprünglich eine Entwicklung von IBM, ist aber auch von der **ANSI** zertifiziert. Die Organisation **American National Standards Institute (ANSI)** koordiniert US-Standards mit internationalen Standards. ANSI zertifiziert auch Standards, die von anderen Standardisierungsinstituten, Regierungseinrichtungen, Firmen oder Anwendergruppen erarbeitet wurden. IBM setzte CRC-16-IBM erstmals bei seinem Protokoll **Synchronous Data Link Control (SDLC)** ein, einem Vorgänger vom und Grundlage für das erweiterte HDLC-Protokoll.

CRC-16-CCITT ist ein vom **CCITT** (**Comité Consultative de Téléphonique et Télégraphique)** standardisiertes Verfahren. CCITT war eine Vorgängerorganisation der **ITU (International Telecommunication Union)**, einem UN-Gremium zur Vereinheitlichung und Normierung von Telekommunikationsstandards. CRC-16-CCITT wird z. B. eingesetzt beim **High-Level Data Link Control (HDLC)**, einem Protokoll für die Sicherungsschicht im OSI-Referenzmodell. Es wurde von der **ISO** (**International Organization for Standardization**) entwickelt und beruht auf dem IBM-Vorgänger **SDLC**. Eine weitere Anwendung von CRC-16-CCITT ist **x.25**, eine von der ITU standardisierte Protokollfamilie für großräumige Computernetze (**Wide Area Network**, **WAN**) über das Telefonnetz. Der Standard definiert die Bitübertragungsschicht, die Sicherungsschicht und die Vermittlungsschicht, d. h. Schichten 1 bis 3 des OSI-Modells. In den 1980ern waren x.25-Netze weit verbreitet, sie sind zwar immer noch in Benutzung, jedoch mit stark sinkender Bedeutung.

6.6.3 Anwendung: CRC bei Schnittstellen

Auch bei der Datenübertragung via Schnittstelle zwischen Computern, Netzwerken und Peripheriegeräten ist die Fehlererkennung ein wichtiges Thema.

Standardschnittstelle USB

Jedem bekannt ist sicher **USB (Universal Serial Bus)**, ein Industriestandard aus den 1990ern, der das Kommunikationsprotokoll zwischen Rechner und Peripheriegeräten (Drucker, Maus, externer Speicher etc.) festlegt. Die Struktur der Schnittstellendatenpakete ist dabei wie folgt.

Jedes Paket hat einen **PID** (Packet Identifier), der 4 Bits an Information umfasst. Zur Fehlersicherung wird dieser – aber mit invertierten Bits – jeweils wiederholt mitgesendet. Damit hat die PID insgesamt 8 Bits. Es gibt einerseits die **Data Packets** (Datenpakete), die die zu übertragenden Inhalte übermitteln (z. B. Druckdatei eines Dokuments). Ihr Umfang ist variabel mit bis zu 8192 Datenbits, sie werden mit CRC-16-IBM geschützt.

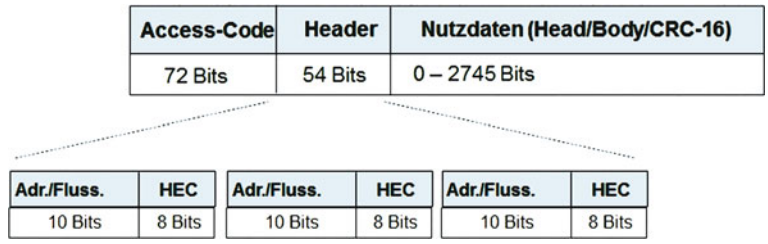


Abb. 6.8 Genereller Aufbau eines Bluetoothdatenpakets

Andererseits verwendet USB sog. **Token Packets** (Adresspakete), die die Adressen und Endpunkte für die Informationen tragen (z. B. Drucker-IP). Dabei handelt es sich um 11 Bits. Da diese Pakete also wenige Bits umfassen, nutzt man hierfür das Polynom CRC-5-USB.

Funkschnittstelle Bluetooth

Bluetooth ist ein in den 1990ern durch die **Bluetooth Special Interest Group** (**SIG**) entwickelter Industriestandard für die Datenübertragung zwischen Geräten über kurze Distanz per Funktechnik. Der Name Bluetooth leitet sich dabei vom dänischen König Harald Blauzahn ab. Bluetooth bildet eine Schnittstelle, über die sowohl mobile Kleingeräte wie Mobiltelefone und Tablets als auch Computer und Peripheriegeräte miteinander kommunizieren können. Hauptzweck von Bluetooth ist das Ersetzen von Kabelverbindungen zwischen Geräten. In Abb. 6.8 ist der generelle Aufbau eines Bluetoothdatenpakets dargestellt.

Der Access-Code enthält Daten zur Netzerkennung und Synchronisation. Der Paket-Header umfasst u. a. Adressdaten und Daten zur Flusssteuerung (10 Bits) sowie eine FCS von 8 Bits (**HEC**, **Header Error Control**), generiert vom CRC-Polynom CRC-8-Bluetooth. Zur besonderen Sicherheit werden diese 18 Bits dreimal wiederholt zu insgesamt 54 Bits, was einem zusätzlichen FEC-Wiederholungscode entspricht. Der Nutzdatenbereich (Payload) hat grundsätzlich variable Länge von bis zu 2745 Bits und gliedert sich weiter in einen Payload-Head, Payload-Body und eine FCS. Diese wird generiert von CRC-16-CCITT, erzeugt also 16 CRC-Bits. Auch hier wird zur besonderen Sicherheit der gesamte Nutzdatenbereich (Payload) noch einem FEC-Verfahren unterworfen. Genauer gesagt werden jeweils 10 Bits in ein Codewort von 15 Bits codiert, wobei der zyklische Code vom Polynom CRC-5-Bluetooth erzeugt wird und damit die Parameter [15, 10] hat.

Kabelloses Funknetz WLAN

WLAN (**Wireless Local Area Network** oder auch **Wi-Fi**) bezeichnet ein lokales Funknetz, das auf dem Standard **IEEE 802.11** beruht und mit dem man sich mit einem Laptop oder Smartphone kabellos in ein Netzwerk (z. B. Internet) einwählen kann. In dieser Konstellation spielt WLAN eher die Rolle einer Schnittstelle. Im **SoHo-Bereich** (Small Office,

Home Office) wird es aber auch eigenständig genutzt, um Geräte (Computer, Drucker etc.) drahtlos miteinander zu vernetzen. Ein MAC-Frame im WLAN besteht generell aus einem Header (30 Bytes), dem Frame-Body (Nutzdaten von bis zu 2312 Bytes) und einer CRC-Folge von 4 Bytes (= 32 Bits), die vom CRC-32-Polynom erzeugt wird.

On-Board-Diagnose
Der Schnittstellenstandard **SAE-J1850** ermöglicht den Zugriff auf Fahrzeugnetzwerke zur On-Board-Diagnose. Aus dieser US-Entwicklung stammt ursprünglich das Polynom CRC-8-SAE-J1850. Ein SAE-J1850-Datenpaket besteht dabei aus einem Header (bis zu 3 Bytes), den Nutzdaten (bis zu 8 Bytes) und der CRC-Folge (1 Byte).

6.6.4 Anwendung: Prozessleittechnik – das Protokoll Modbus

Eine der wichtigsten Anwendungen von CRC-16-IBM ist **Modbus**. Es handelt sich dabei um ein Kommunikationsprotokoll

- zur speicherprogrammierbaren Steuerung **SPS** (**Programmable Logic Controller, PLC**) von Geräten,
- die ihrerseits zur Steuerung oder Regelung einer Maschine oder Anlage eingesetzt und entsprechend programmiert werden.

Dieses Protokoll aus dem Jahr 1979 ursprünglich von Firma Modicon (heute Schneider Electric) ist mittlerweile de facto Standard in der **industriellen Steuerungs- und Regelungstechnik**. Wir bewegen uns damit also in einer ganz anderen Welt – weg von der alltäglichen Internet- und Bürokommunikation, hin zu (hoch) sicherheitsrelevanten Anwendungen in Raffinerien, großchemischen Anlagen und Kraftwerken. Ein Modbus-Datenpaket heißt **Telegramm** und umfasst die Adresse (1 Byte), einen Funktionscode (1 Byte) und n Datenbytes. Zur Sicherung werden 2 Bytes (= 16 Bits) mittels des Polynoms CRC-16-IBM angehängt.

6.6.5 Anwendung: LAN und Ethernet

Ethernet ist ein Protokoll der Bitübertragungsschicht und der Sicherungsschicht (d. h. Schichten 1 und 2) im OSI-Referenzmodell für kabelgebundene Datennetze – gemäß Standard **IEEE 802.3** –, welches ursprünglich nur für lokale Datennetze (**Local Area Networks**, **LAN**) gedacht war und daher auch als LAN-Technik bezeichnet wird. Es ermöglicht den Datenaustausch in Form von Datenframes zwischen den in einem lokalen Netz (LAN) angeschlossenen Geräten. In seiner ursprünglichen Form erstreckte sich ein LAN dabei nur über ein Gebäude. Ethernet über Glasfaser hat aber mittlerweile eine Reichweite von 10 km und mehr (**MAN**, **Metropolitan Area Network**). Auch beim

Ethernet wird zur Erzeugung einer FCS das Polynom CRC-32 eingesetzt. Mit CRC-32 kann jeder einzelne Ethernet-Datenframe mit 32 Paritätsbits versehen werden. Ein Datenframe des Ethernets umfasst nämlich neben einigen Steuerdaten, wie z. B. Quell- und Ziel-MAC-Adresse, einen Nutzdatenanteil von bis zu 1500 Bytes, der zur Datensicherheit um 4 Bytes (= 32 Bits) mittels CRC-32 erweitert wird.

Die Theorie lautet: Wenn bei der Überprüfung mittels Division durch das CRC-Polynom der Rest 0 herauskommt, wurde kein Fehler erkannt. Beim Ethernet – und bei übrigens den meisten anderen Netzwerkprotokollen auch – geht man leicht modifiziert vor, um das sog. **Nullproblem** zu umgehen. Es ist nämlich problematisch, wenn der CRC-Wert fast nur aus Nullen besteht, da bei der Datenübertragung oft zusätzliche Nullen als Füllbits ergänzt werden. Dieses Problem wird beim Ethernet dadurch vermieden, dass man die ersten 32 Bits in der MAC-Adresse eines Datenpakets sowie auch die 32 CRC-Bits invertiert, d. h das Einerkomplement bildet. Man kann sich vorstellen, dass dann bei der Prüfung durch den Empfänger auch bei einem korrekten Codewort nicht die 0 herauskommt. Es kommt stattdessen die **Magic Number** genannte Hexadezimalzahl 0xC704DD7B heraus. Das Präfix „0x“ kennzeichnet dabei Hexadezimalzahlen, also Zahlendarstellungen bzgl. der Basis $2^4 = 16$. Die restlichen acht Zeichen (Ziffern und Buchstaben) beschreiben die 32 Bits des Rests bei Division durch das CRC-Polynom hexadezimal.

Wie häufig muss aber ein Empfänger in der Praxis per ARQ-Signal die Information wiederholt anfordern? Das könnte ja den gesamten Datenfluss zum Erliegen bringen. Zur Beruhigung sei gesagt, dass die Bitfehlerwahrscheinlichkeit im Ethernet etwa bei 10^{-10} liegt.

6.6.6 Anwendung: Digitale Telefonie mit ISDN

In diesem Zusammenhang wollen wir auch noch kurz auf den Standard **ISDN (Integrated Services Digital Network)** zu sprechen kommen, der aus den 1980ern stammt. Er war der erste internationale Standard für digitale kabelgebundene Telekommunikation – vor allem für Telefonie, aber auch für andere Dienste wie Telex und Datex. Ein Datenpaket des Breitband-ISDN (**ATM**, **Asynchronous Transfer Mode**) besteht aus 5 Bytes als Kopf (Header) und 48 Bytes für die eigentlichen Informationsdaten. Dabei ist nur der Header durch das CRC-Polynom CRC-8-CCITT geschützt, mittels dessen an die eigentlichen 4 Bytes des Kopfs ein weiteres Byte (= 8 Bits) angehängt wird. Es handelt sich dabei also wieder um **HEC** (**Header Error Control**). Auf die neuere Generation **DSL** gehen wir in Abschn. 9.6 ein.

6.6.7 Anwendung: Frame Checking Sequence im Internet TCP/IP

Das **IP** (**Internet Protocol)** ist das Grundelement der Internetdatenkommunikation, angesiedelt in der Vermittlungsschicht, also der 3. Schicht im OSI-Modell. Das **Transmission Control Protocol** (**TCP**) ist ein Netzwerkprotokoll der Transportschicht, d. h. der 4. Schicht im OSI-Modell, das definiert, auf welche Art und Weise Daten zwischen Computern ausgetauscht werden sollen. Nahezu sämtliche aktuellen Betriebssysteme moderner Computer beherrschen TCP und nutzen es für den Datenaustausch mit anderen Rechnern. TCP setzt in den meisten Fällen auf IP auf, weshalb häufig – aber nicht ganz korrekt – auch vom Gesamtprotokoll **TCP/IP** die Rede ist. TCP wird als fast ausschließliches Transportmedium für WWW, E-Mail und viele andere populäre Netzdienste verwendet. Die Schichten 1 und 2 des OSI-Modells sind bei TCP/IP nicht festgelegt. Die Internetprotokolle wurden nämlich mit dem Ziel entwickelt, verschiedene Subnetze zusammenzuschließen. Hier können unterschiedliche Techniken zur Datenübertragung von Punkt zu Punkt integriert werden (wie z. B. Ethernet, HDLC).

Ein TCP/IP-Paket besteht aus einem Header mit Informationen über Quelle, Ziel, Status etc. sowie einem Datenteil, der wiederum ein eigenes Protokoll, in diesem Fall TCP, enthält. Eine Prüfsequenz sichert hier ausschließlich den Kopfdatenbereich und wir kommen damit auch zum enttäuschenden Teil der Geschichte. Das Internet mit seinem IP-Protokoll nutzt kein CRC, wohl aber eine einfachere Form einer FCS. Der Grund ist einleuchtend. Bei IP muss nämlich die Prüfsequenz bei jedem Hop (Netzknotenpunkt) neu berechnet werden. Das ist zu aufwendig bei zuverlässigeren CRC-Prüfverfahren. Hier ist also der Algorithmus für das einfachere FCS-Verfahren.

- Der Sender fasst jeweils zwei benachbarte Bytes zu 16-Bit-Blöcken (einem *Wort*) zusammen.
- Er addiert danach zwei benachbarte Worte wie Integer mit Übertrag.
- Kommt ein Übertrag zustande, so bildet er zusätzlich das 16-Bit-Wort $(0, 0, \ldots, 0, 1)$ und addiert dies auf den 16-Bit-Anteil der oben gebildeten Summe, auch wiederum mit Übertrag.
- Er wiederholt dies für die so gebildete Summe und das nächste Wort, bis das Ende des Informationsstrings erreicht ist.
- Von diesem 16-Bit-Tupel bildet er das Einerkomplement (d. h. er vertauscht 0 und 1).
- Schließlich sendet er das Resultat als FCS mit der Dateneinheit.
- Der Empfänger führt dieselben Additionen aus und addiert schließlich das Ergebnis bitweise auf die FCS.
- Ergibt sich dort nicht $(0, \ldots, 0)$, so ist ein Fehler aufgetreten.

Dies machen wir uns nochmal an einem konkreten Beispiel in Abb. 6.9 klar mit nur zwei Worten als Informationsstring. Die zwei 16-Bit-Blöcke werden also zuerst addiert – wie Integer mit Übertrag. Es ergibt sich vorn der Übertrag „1“. Daher wird $(0, 0, \ldots, 0, 1)$

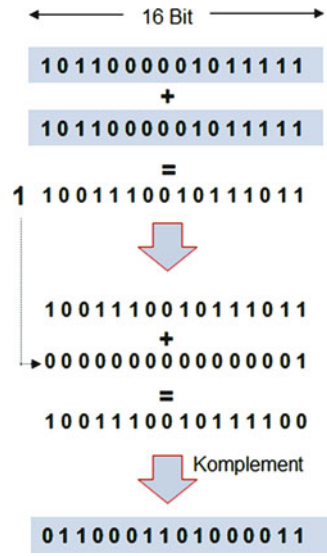


Abb. 6.9 Beispiel für eine FCS im Internet IP

dazu addiert – wiederum mit Übertrag. Daraus wird das Einerkomplement gebildet und als FCS gesendet.

6.6.8 Anwendung: PGP (Pretty Good Privacy)

Im Zusammenhang mit dem Internet nochmals ein kleiner Ausflug in die Kryptografie, genauer gesagt zum Chiffrierverfahren **PGP (Pretty Good Privacy)**, das erstmalig von **Phil Zimmermann** (geb. 1954 in New Jersey/USA) im Jahr 1991 veröffentlicht wurde. Es enthält als Bausteine sowohl

- den symmetrischen **Triple-DES**-Algorithmus (**Data Encryption Standard**) zur eigentlichen Datenverschlüsselung als auch
- die asymmetrischen **RSA-**(**Rivest-Shamir-Adleman-**) und **ElGamal**-Verfahren zur Schlüsseletablierung und zur Signatur.

PGP wird insbesondere gerne zur Verschlüsselung und Authentifizierung von E-Mails im Internet genutzt. Die PGP-Datenstruktur beruht zum einen auf dem **Base64**-Schema, bei dem jeweils drei Bytes des chiffrierten Bitstroms (also 24 Bits) in vier 6-Bit-Blöcke aufgeteilt werden. Jeder dieser 6-Bit-Blöcke bildet eine Zahl von 0 bis 63, die anhand einer Umsetzungstabelle in die druckbaren Zeichen *Großbuchstaben, Kleinbuchstaben, Ziffern* sowie + und / umgewandelt werden. Bei PGP wird zur Fehlererkennung die Erweiterung **Radix-64** eingesetzt, bei der einem Base64-Text eine 24-Bit-FCS mittels des Polynoms CRC-24-Q angehängt wird. Dies ist in Abb. 6.10 verdeutlicht.

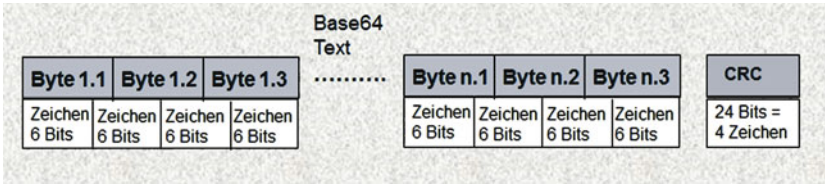


Abb. 6.10 Base64-/Radix-64-Datenstruktur von PGP

6.6.9 Anwendung: Speicherchips – SD-Karten

Wir kommen nun zu einem anderen Thema, nämlich der Speicherung von Daten auf Chipkarten. Eine **Multimedia Card (MMC)** ist ein digitales Speichermedium für beispielsweise Digitalkameras, MP3-Player, GPS-Navigationsgeräte und Handys. Der MMC-Standard wurde 1997 von Siemens zusammen mit SanDisk entwickelt. MMCs besitzen mehrere kleine elektrische Bauelemente, die über einen integrierten Controller angesteuert werden. Die **SD-Karte** (**Secure Digital Memory Card**) stellt eine Weiterentwicklung der Firma SanDisk auf Basis des älteren MMC-Standards dar. Sie zählt wie auch USB-Sticks zu den **Flash-Speichern**.

Um mit einer SD-Karte zu arbeiten, schickt man unterschiedliche Kommandos an die Karte und erhält darauf Antworten. Eventuell sendet oder empfängt man danach ein oder mehrere Datenpakete. Generell kommunizieren SD-Karten über das SD-Busprotokoll. Die Übertragung läuft ab auf einer Kommandoleitung, auf der die Karte Befehle entgegennimmt und gegebenenfalls darauf antwortet, und mehreren Datenleitungen, auf denen die Datenpakete übertragen und bestätigt werden. Alle Kommandos und Datenpakete enthalten eine CRC-Prüfsequenz, um eine korrekte Übertragung sicherzustellen. Wenn die Karte ein Kommando oder Datenpaket mit falscher CRC bekommt, wird dieses abgelehnt und eine Fehlermeldung wird zurückgesendet.

- Kommandos werden als Folge von 40 Bits dargestellt. Zur Sicherung wird hierbei das CRC-7-Polynom verwendet, man hängt also eine siebenstellige FCS an.
- Nutzdaten werden beim Transfer in Paketen mit maximal 2048 Bytes (d. h. 16.384 Bits) gesendet. Für deren CRC-Codierung wird das Polynom CRC-16-CCITT genutzt, welches den Nutzdaten eine FCS von 16 Bits anhängt.

6.6.10 Anwendung: Datenkompression im .zip-Format

Bei der Datenkompression im **.zip-Format** werden die komprimierten Dateien ebenfalls mit CRC-32 geschützt. Wenn nämlich eine Datei zu einem Archiv hinzugefügt wird, berechnet das Kompressionsprogramm (z. B. Winzip oder 7-Zip) deren CRC-Wert und hinterlegt diesen zusammen mit anderen Informationen, wie z. B. der ursprünglichen und

komprimierten Dateigröße, im sog. Local Header der Archivdatei. Beim Extrahieren berechnet das Programm den CRC-Wert der wiederhergestellten Datei und vergleicht ihn mit dem im Archiv gespeicherten Wert. Weichen diese beiden voneinander ab, so ist die extrahierte Datei nicht mit der Originaldatei identisch und es erfolgt die Ausgabe einer CRC-Fehlermeldung. Hier wird also – wie auch bei PGP – CRC für Bitfolgen *beliebiger* Länge gebildet.

Zyklische Reed-Solomon- und BCH-Codes

7

Primitive Einheitswurzeln Um weiter voranzukommen, müssen wir jetzt noch ein klein wenig mehr wissen über endliche Körper. Es handelt sich um die Tatsache, dass jeder endliche Körper eine **primitive Einheitswurzel** besitzt, d. h. ein Element, bei dem man durch Potenzieren alle Körperelemente ungleich 0 erhält. Wir überprüfen dies zunächst anhand unserer kleinen endlichen Körper. Mithilfe dieses Werkzeugs sehen wir dann sehr schnell, dass auch **binäre Hamming-Codes zyklische Codes** sind. Und – wie zu vermuten – auch **Reed-Solomon-Codes** sind **zyklisch**: Wir geben ganz konkret ein Generatorpolynom an.

BCH-Codes und BCH-Schranke Fast gleichzeitig zu den Reed-Solomon-Codes wurden 1959/60 auch die **BCH-Codes** veröffentlicht – benannt nach **Alexis Hocquenghem, Raj Chandra Bose** und **Kumar Ray-Chaudhuri** –, von der Bedeutung her sicher vergleichbar mit der aufsehenerregenden Publikation von Reed und Solomon. BCH-Codes werden als zyklische Codes definiert, wobei wir uns besonders um den wichtigsten Spezialfall kümmern wollen – **primitive BCH-Codes im engeren Sinne**. Für sie jedenfalls gilt die sog. **BCH-Schranke**, eine untere Abschätzung für den Minimalabstand. Außerdem werden uns zwei Dinge bewusst, nämlich erstens: **Reed-Solomon-Codes** sind ein **Spezialfall von BCH-Codes**, und zweitens: BCH-Codes erlauben es, wieder **binär** zu arbeiten, wohingegen Reed-Solomon-Codes größere Körper als Alphabet benötigen. Auch hier machen wir uns wieder anhand von Beispielen mit der Materie vertraut, unter anderem auch am Beispiel eines **Pagers**.

QR-Code – Quick Response Jetzt haben wir wieder genügend konzeptionelles Material gesammelt, um uns erneut recht detailliert in ein Anwendungsbeispiel stürzen zu können – den **2-D-Barcode Q(uick)R(esponse)**. Wenn man nämlich einen solchen Barcode fotografisch erfasst, dann hält man das Handy sicher nicht im optimalen Winkel und möglicherweise ist der Code auch ein wenig geknickt. Trotzdem erwartet man, dass die Information korrekt ausgelesen wird. Und dafür ist natürlich auch Sorge getragen. Je nach Sensibilität der Anwendung ist ein QR-Code in vier verschiedenen **Fehlerkorrekturstu-**

O. Manz, *Fehlerkorrigierende Codes*, DOI 10.1007/978-3-658-14652-8_7

fen verfügbar. Bei jeder dieser Stufen kommen unterschiedliche **Reed-Solomon-Codes** und zusätzlich ein einheitlicher **BCH-Code** zum Einsatz. Wir gehen die Struktur des QR-Codes im Einzelnen durch.

PGZ-Decodierung Wie nicht anders zu erwarten: Wir müssen uns natürlich auch wieder über die Decodierung von Reed-Solomon-Codes und BCH-Codes unterhalten. Fast gleichzeitig mit deren Publikation haben 1960/61 **Westley Peterson**, **Daniel Gorenstein** und **Neal Zierler** ein auf diese Codes zugeschnittenes Decodierverfahren angegeben, das unter dem Namen **PGZ-Decodierung** bekannt ist. Es verwendet das sog. **Fehlerortungspolynom**, um dessen Bestimmung sich der Algorithmus im Wesentlichen dreht. Wir machen uns am Beispiel der Reed-Solomon-Codes klar, warum der Algorithmus funktioniert. Dann formulieren wir das Decodierverfahren Schritt für Schritt – auch für BCH-Codes – und üben es an einigen kleinen Beispielen.

Berlekamp-Massey-Algorithmus Leider müssen wir noch etwas nachtragen. Wir haben nämlich noch nicht erwähnt, dass das PGZ-Verfahren für viele praxisrelevante Anwendungen einfach zu langsam ist. Wenn man nämlich das beim PGZ-Algorithmus notwendige Lösen eines speziellen Gleichungssystems mit allgemeinen Methoden (z. B. Gauß'sches Eliminationsverfahren) angeht, so erleidet man beim Antwortzeitverhalten Schiffbruch. Dies änderte sich erst 1965, als **Elwyn Berlekamp** einen schnelleren Algorithmus angab, und besonders 1969, als **James Lee Massey** diesen mit einer Schieberegisterschaltung in Verbindung brachte. Wir machen uns den sog. **Berlekamp-Massey-Algorithmus** klar, auch anhand einiger Beispiele. Schließlich betrachten wir noch eine weitere – eher kleine – Beschleunigung des ursprünglichen PGZ-Algorithmus, nämlich die Auswertung des Fehlerortungspolynoms mittels der sog. **Chien Search**.

Justesen-Codes, Goppa-Codes und Euklid-Decodierung Wir vermerken noch eine weitere Verallgemeinerung der Reed-Solomon-Codes, die 1972 publizierten **Justesen-Codes**, die allerdings weniger Bedeutung erfahren haben. Viel wichtiger sind hingegen die **Goppa-Codes** – aber auch viel schwieriger vom Verständnis her. Wir machen uns die Konstruktion der von **Valery Goppa** 1970 publizierten **klassischen Goppa-Codes** klar – nämlich als Verallgemeinerung von BCH-Codes. Die Bedeutung der Goppa-Codes hängt in erster Linie zusammen mit der durch sie initiierten Anwendung von Methoden aus der **algebraischen Geometrie**. Bei dieser Gelegenheit lernen wir noch ein weiteres Decodierverfahren kennen, welches auf dem **euklidischen Algorithmus** beruht und das nicht nur allgemein bei Goppa-Codes, sondern vor allem auch bei BCH- und Reed-Solomon-Codes Anwendung findet. Hierfür rechnen wir wieder konkrete Beispiele durch.

7.1 Primitive Einheitswurzeln

Wir sind bislang ja so vorgegangen, dass wir uns anfänglich völlig ohne algebraische Hilfsmittel in die Codierungstheorie *gestürzt* haben und dann stückweise immer ein klein wenig mehr Algebra verwendet haben. Wir haben dabei auch festgestellt, dass sich einiges – wenn auch nicht alles – in Analogie zu $\mathbb{R}$ verstehen lässt. Jetzt müssen wir einen weiteren Schritt gehen und uns über Einheitswurzeln unterhalten.

7.1.1 Primitive Einheitswurzeln in endlichen Körpern

Sei K ein Körper und n eine natürliche Zahl. Die Nullstellen des Polynoms $x^n - 1$ nennt man ***n*. Einheitswurzeln**.

In diesem Zusammenhang zwischendurch wieder eine kleine Analogieüberlegung für alle diejenigen, die schon etwas von komplexen Zahlen $\mathbb{C}$ gehört haben. Wenn man das Polynom $x^n - 1$ nur über $\mathbb{R}$ betrachtet, dann ist eine n. Einheitswurzel entweder nur 1, falls n ungerade ist, oder 1 oder -1, wenn n gerade ist. Um aber alle Nullstellen von $x^n - 1$ zu erhalten, muss man in einen größeren Körper gehen, den der **komplexen Zahlen** $\mathbb{C}$. Dort nämlich gibt es genau n verschiedene Nullstellen, die alle auf dem Einheitskreis in der sog. **komplexen Zahlenebene** liegen und jeweils um einen Winkel von $360°/n$ gedreht sind. Wenn man also die erste n. Einheitswurzel in $\mathbb{C}$ nimmt (die den Winkel $360°/n$ mit der positiven reellen Achse bildet), so kann man durch wiederholtes Weiterdrehen um jeweils $360°/n$ alle anderen n. Einheitswurzeln erhalten. Abb. 7.1 zeigt ein Beispiel für $n = 8$.

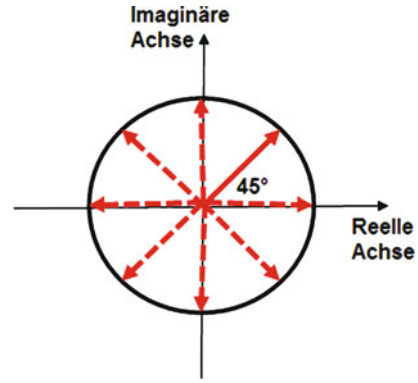


Abb. 7.1 Einheitswurzeln in der komplexen Zahlenebene

Für endliche Körper K verhält sich das noch besser.

Sei dazu $|K| = q$, $K^* = K/\{0\}$ und $n = q - 1$. Dann sind *alle* Elemente aus K^* Nullstellen des Polynoms $x^n - 1$, also n. Einheitswurzeln. Außerdem gibt es (mindestens) ein Element $\beta \in K^*$ mit $K^* = \{\beta^0 = 1, \beta, \beta^2, \ldots, \beta^{n-1}\}$ und $\beta^n = 1$. Das Element β heißt **primitive *n*. Einheitswurzel**, es ist nämlich Nullstelle des Polynoms $x^n - 1$, aber keine Nullstelle von $x^m - 1$ für alle $m < n$. Außerdem lässt sich das Polynom $x^n - 1$ faktorisieren in $x^n - 1 = (x - 1)(x - \beta)(x - \beta^2) \ldots (x - \beta^{n-1}) \in K[x]$. In Worten lautet die Kernaussage: **Es gibt stets eine sog. primitive Einheitswurzel in *K**, mit der man durch Potenzieren den ganzen Körper *K* ohne die 0 erhält**.

Wir können und wollen dieses Resultat hier nicht herleiten. Formale Beweise findet man z. B. in den Büchern von Willems [Wil2] und van Lint [vLi82, vLi99].

Dennoch können wir die Aussage an einigen endlichen Körpern überprüfen.

- $K = \mathbb{Z}_3$ und $\beta = -1$.
  Dann ist $K^* = \{\beta^0 = 1, \beta = -1\}$ und $\beta^2 = 1$.
- $K = \mathbb{Z}_5$ und $\beta = 2$.
  Dann ist $K^* = \{\beta^0 = 1, \beta = 2, \beta^2 = 4, \beta^3 = 3\}$ und $\beta^4 = 1$.
- $K = \mathbb{Z}_7$ und $\beta = 3$.
  Dann ist $K^* = \{\beta^0 = 1, \beta = 3, \beta^2 = 2, \beta^3 = 6, \beta^4 = 4, \beta^5 = 5\}$ und $\beta^6 = 1$.
- $K = \mathbb{Z}_{11}$ und $\beta = 2$.
  Dann ist $K^* = \{\beta^0 = 1, \beta = 2, \beta^2 = 4, \beta^3 = 8, \beta^4 = 5, \beta^5 = 10, \beta^6 = 9, \beta^7 = 7, \beta^8 = 3, \beta^9 = 6\}$ und $\beta^{10} = 1$.

Für die folgenden Körper müssen wir unsere Multiplikationstafeln aus Abschn. 5.1 verwenden.

- $|K| = 2^2$ und $\beta = t$ mit $t^2 = t + 1$.
  Dann ist $K^* = \{\beta^0 = 1, \beta = t, \beta^2 = t + 1\}$ und $\beta^3 = 1$.
- $|K| = 2^3$ und $\beta = t$ mit $t^3 = t + 1$.
  Dann ist $K^* = \{\beta^0 = 1, \beta = t, \beta^2 = t^2, \beta^3 = t+1, \beta^4 = t^2+t, \beta^5 = t^2+t+1, \beta^6 = t^2 + 1\}$ und $\beta^7 = 1$.
- $|K| = 2^4$ und $\beta = t$ mit $t^4 = t + 1$.
  Dann ist $K^* = \{\beta^0 = 1, \beta = t, \beta^2 = t^2, \beta^3 = t^3, \beta^4 = t + 1, \beta^5 = t^2 + t, \beta^6 = t^3 + t^2, \beta^7 = t^3 + t + 1, \beta^8 = t^2 + 1, \beta^9 = t^3 + t, \beta^{10} = t^2 + t + 1, \beta^{11} = t^3 + t^2 + t, \beta^{12} = t^3 + t^2 + t + 1, \beta^{13} = t^3 + t^2 + 1, \beta^{14} = t^3 + 1\}$ und $\beta^{15} = 1$.
- $|K| = 3^2$ und $\beta = t + 1$ mit $t^2 = -1$.
  Dann ist $K^* = \{\beta^0 = 1, \beta = t + 1, \beta^2 = -t, \beta^3 = -t + 1, \beta^4 = -1, \beta^5 = -t - 1, \beta^6 = t, \beta^7 = t - 1\}$ und $\beta^8 = 1$.

Mutige können das Ganze auch noch mit $|K| = 2^5$ oder sogar $|K| = 2^8$ durchführen. Bei $|K| = 2^5$ etwa ist $\beta = t$ mit $t^5 = t^2 + 1$ eine primitive 31. Einheitswurzel. Wir üben das Rechnen mit primitiven Einheitswurzeln aber auch gleich noch anhand folgender Aussage, die wir auf Basis der Beispiele aus Abschn. 6.3 schon vermuten konnten.

7.1.2 Hamming-Codes – *zyklisch reloaded*

Auch binäre Hamming-Codes $Ham_2(k)$ sind zyklische Codes, d. h., eine der äquivalenten Formen des Codes ist zyklisch.

Um das einzusehen, betrachten wir den Körper K mit $q = 2^k$ Elementen. Wir wissen auch, dass K ein Vektorraum der Dimension k über dem Grundkörper $\mathbb{Z}_2$ ist. Aber wir wissen jetzt noch mehr, nämlich dass wir uns eine primitive n. Einheitswurzel β wählen können, wobei $n = q - 1$ ist, und es gilt dann $K^* = \{\beta^0 = 1, \beta, \ldots, \beta^{n-1}\}$. Daraus konstruieren wir uns eine Matrix

$$M_0 = \begin{pmatrix} (\beta^0)_0 & \beta_0 & (\beta^2)_0 & \ldots & (\beta^{n-1})_0 \\ \vdots & \vdots & \vdots & & \vdots \\ (\beta^0)_{k-1} & \beta_{k-1} & (\beta^2)_{k-1} & \ldots & (\beta^{n-1})_{k-1} \end{pmatrix},$$

indem wir in die Spalten der Reihenfolge nach die β^i schreiben, aber als k-Tupel über $\mathbb{Z}_2$ aufgefasst. Das können wir machen, weil ja K ein k-dimensionaler Vektorraum über $\mathbb{Z}_2$ ist und sich jedes β^i daher schreiben lässt als $\beta^i = (\beta^i)_0 t^0 + \ldots + (\beta^i)_{k-1} t^{k-1}$ mit der $\mathbb{Z}_2$-Basis $\{t^0, t^1, \ldots, t^{k-1}\}$ von K. Die $n = 2^k - 1$ Spalten der Matrix M_0 sind dann aber gerade alle Vektoren $\neq 0$ in $(\mathbb{Z}_2)^k$. Das wiederum ist aber genau das Konstruktionsprinzip der Matrix M, die wir zur Konstruktion der Hamming-Codes verwendet haben. Da es dabei nicht auf die Reihenfolge der Spalten ankam, wird durch M_0 als Kontrollmatrix eine äquivalente Form C des Hamming-Codes $Ham_2(k)$ definiert. Es ist $c = (c_0, \ldots, c_{n-1}) \in (\mathbb{Z}_2)^n$ genau dann in C, wenn der Vektor c Skalarprodukt $\langle\, , \rangle = 0$ mit allen Zeilen von M_0 hat. Dies lässt sich aber auch schreiben als $c_0\beta^0 + c_1\beta^1 + \ldots + c_{n-1}\beta^{n-1} = 0$. Multipliziert man dies mit β, so gilt wegen $\beta^n = 1$ auch die Gleichung $0 = (c_0\beta^0 + c_1\beta^1 + \ldots + c_{n-1}\beta^{n-1})\beta = c_{n-1}\beta^0 + c_0\beta + c_1\beta^2 + \ldots + c_{n-2}\beta^{n-1}$. Daher ist mit $c = (c_0, \ldots, c_{n-1})$ auch das Wort $(c_{n-1}, c_0, c_1, \ldots, c_{n-2})$ in C und C ist damit zyklisch.

Ein Generatorpolynom haben wir für $Ham_2(k)$ nicht explizit konstruiert. Genau das werden wir aber jetzt für Reed-Solomon-Codes tun.

7.1.3 Zyklische Reed-Solomon-Codes

Auch Reed-Solomon-Codes $RS_q(d)$ mit den Parametern $[q-1, q-d, d]_q$ sind – bis auf Äquivalenz – zyklische Codes. Mit den Bezeichnungen $n = q - 1$ und β

eine primitive n. Einheitswurzel im Körper K mit q Elementen gilt genauer, dass $RS_q(d)$ folgendes Generator- und Kontrollpolynom sowie folgende Generator- und Kontrollmatrix besitzt.

$$g(x) = (x-\beta)(x-\beta^2)\dots(x-\beta^{d-1})$$
$$h(x) = (x-\beta^d)(x-\beta^{d+1})\dots(x-\beta^n)$$
$$G = \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \beta & \beta^2 & \dots & \beta^{n-1} \\ 1 & \beta^2 & (\beta^2)^2 & \dots & (\beta^{n-1})^2 \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & \beta^{n-d} & (\beta^2)^{n-d} & \dots & (\beta^{n-1})^{n-d} \end{pmatrix}$$
$$H = \begin{pmatrix} 1 & \beta & \beta^2 & \dots & \beta^{n-1} \\ 1 & \beta^2 & (\beta^2)^2 & \dots & (\beta^{n-1})^2 \\ 1 & \beta^3 & (\beta^2)^3 & \dots & (\beta^{n-1})^3 \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & \beta^{d-1} & (\beta^2)^{d-1} & \dots & (\beta^{n-1})^{d-1} \end{pmatrix}$$

Hierzu haben wir nun einiges nachzuweisen.

G ist zunächst genau die Generatormatrix, die wir bereits in Abschn. 5.2 konstruiert hatten. Wir haben für die $\alpha_1, \dots, \alpha_n$ nur die Potenzen $1, \beta, \dots, \beta^{n-1}$ der primitiven Einheitswurzel β eingesetzt, und zwar in dieser Reihenfolge.

Sobald wir uns überlegt haben, dass $g(x)$ das Generatorpolynom ist, dann ist $h(x)$ wegen $(x^n - 1) = g(x)h(x)$ das Kontrollpolynom.

Wir betrachten nun den zyklischen Code C der Länge n mit Generatorpolynom $g(x)$ und zeigen, dass H eine Kontrollmatrix für C ist. Es ist also $c(x) = c_{n-1}x^{n-1} + \dots + c_1x + c_0$ genau dann in C, wenn $g(x)$ ein Teiler von $c(x)$ ist. Wegen $g(x) = (x-\beta)(x-\beta^2)\dots(x-\beta^{d-1})$ bedeutet dies aber für $i = 1, \dots, d-1$, dass $c(\beta^i) = c_{n-1}(\beta^i)^{n-1} + \dots + c_1(\beta^i) + c_0 = 0$ ist. Also hat – in Tupelschreibweise – das Codewort $c = (c_0, c_1, \dots, c_{n-1})$ Skalarprodukt $\langle\,,\rangle = 0$ mit allen Zeilen der Matrix H. Wenn wir jetzt noch wüssten, dass die $d-1$ Zeilenvektoren von H linear unabhängig sind, so wäre H wegen $\dim(C^\perp) = n - \dim(C) = n - (n - \operatorname{grad}(g(x))) = d-1$ Kontrollmatrix von C.

Warum aber sind die Zeilenvektoren von H linear unabhängig? Zunächst sind sämtliche Elemente β^i der 1. Zeile von H verschieden. Offenbar sind die übrigen Zeilen von H fortlaufende Potenzen der 1. Zeile; Gleiches gilt dann auch für jede quadratische Teilmatrix von H mit $d-1$ Zeilen und Spalten. Man nennt quadratische Matrizen dieser Form **Vandermonde-Matrizen**. Diese spielen eine entscheidende Rolle, wie wir auch noch spä-

ter sehen werden, da man von ihnen stets weiß, dass ihre Zeilen- und Spaltenvektoren linear unabhängig sind.

Um dies formal herzuleiten, benötigt man etwas Matrizenrechnung. Man kann nämlich zeigen, dass Vandermonde-Matrizen stets Determinante $\neq 0$ *besitzen und daher maximalen Rang* $d-1$ *haben.*

Wir zeigen abschließend, dass H auch Kontrollmatrix zu $RS_q(d)$ ist. Dann nämlich ist $C^\perp = RS_q(d)^\perp$ und folglich $C = C^{\perp\perp} = RS_q(d)^{\perp\perp} = RS_q(d)$.

Wir nutzen hierfür wieder die Definition von Reed-Solomon-Codes als Auswertungscode und können ein Codewort c daher schreiben als $c = (f(1), f(\beta), \ldots, f(\beta^{n-1})) \in RS_q(d)$ mit einem Polynom der Gestalt $f(x) = a_{n-d}x^{n-d} + \ldots + a_1 x + a_0$. Wir bilden das Skalarprodukt $\langle\,,\rangle$ von c mit dem i. Zeilenvektor von H und erhalten daher die Gleichung $f(\beta^{n-1})(\beta^{n-1})^i + \ldots + f(\beta)\beta^i + f(1)1 = \sum_j a_j \beta^{(n-1)(j+i)} + \ldots + \sum_j a_j \beta^{j+i} + \sum_j a_j = \sum_j a_j (\beta^{(n-1)(j+i)} + \ldots + \beta^{j+i} + 1)$, wobei die Summen jeweils von $j = 0$ bis $n-d$ laufen. Wegen $i + j \leq (d-1) + (n-d) \leq n-1$ für alle solche i und j wissen wir einerseits $\beta^{j+i} \neq 1$. Andererseits ist aber β^{j+i} eine Nullstelle von $x^n - 1 = (x-1)(x^{n-1} + x^{n-2} + \ldots + x + 1)$ und damit von $x^{n-1} + x^{n-2} + \ldots + x + 1$. Dies bedeutet aber gerade $\beta^{(n-1)(j+i)} + \ldots + \beta^{j+i} + 1 = 0$ und unser Codewort c hat Skalarprodukt $\langle\,,\rangle = 0$ mit allen Zeilen von H. Wegen $\dim(RS_q(d)^\perp) = n - \dim(RS_q(d)) = n - ((n+1) - d) = d - 1$ folgt wieder mit dem Vandermonde-Argument von eben, dass H Kontrollmatrix von $RS_q(d)$ ist.

Das war sicher nicht ganz einfach. Aber als zentrales Ergebnis über Reed-Solomon-Codes und daher auch der Codierungstheorie wollten wir es dennoch ausführlich darstellen. Deshalb sofort wieder ein konkretes Beispiel.

7.1.4 Anwendung: PDF 417

Wir hatten in Abschn. 5.3 bereits die Struktur von PDF 417 besprochen und auch erwähnt, dass die Fehlerbehandlung auf Reed-Solomon-Codes beruht. Jetzt können wir dies präzisieren. Der Barcode PDF 417 lässt nämlich 929 verschiedene Datenzeichen (Symbol Characters) zu, die jeweils im Schnitt mehr als nur ein ASCII-Zeichen repräsentieren. Nun ist 929 eine Primzahl und folglich $K = \mathbb{Z}_{929}$ ein Körper. Wir können daher die Datenzeichen von PDF 417 mit den Elementen im Körper K identifizieren. Außerdem ist – wie man nachrechnen kann – die 3 eine primitive 928. Einheitswurzel in K. Für jede der neun Fehlerkorrekturstufen $0 \leq s \leq 8$ von PDF 417 und für $t = 2^{s+1}$ bildet man nun das Generatorpolynom $g_s(x) = (x-3)(x-3^2)\ldots(x-3^t)$ des Reed-Solomon-Codes $RS_{929}(t+1)$ von Länge $n = 928$ und Minimalabstand $d = t+1$. Dies ist genau der Code, der für PDF 417 mit Fehlerkorrekturstufe s verwendet wird. Mit ihm kann man also eine Folge von bis zu $k = n - d + 1 = 928 - t$ der Information dienenden Datenzeichen um t redundante Zeichen zur Fehlerkorrektur erweitern. Als konkretes Beispiel wählen wir $s = 1$ und folglich $t = 4$. Damit ergibt sich als Generatorpolynom $g_1(x) = (x-3)(x-3^2)(x-3^3)(x-3^4) = x^4 + 809x^3 + 723x^2 + 568x + 522$.

Enthält ein konkreter PDF 417 Barcode weniger als k Informationszeichen, so wird der Reed-Solomon-Code um entsprechend viele Stellen gekürzt.

7.2 BCH-Codes und BCH-Schranke

7.2.1 Was versteht man unter BCH-Codes?

Die Zeit um 1960 muss sehr spannend in der Codierungstheorie gewesen sein. Wir haben ja schon gehört, dass Reed und Solomon ihre nach ihnen benannten Codes im Jahr 1960 publiziert haben, nämlich als Auswertungscodes, wie wir sie anfänglich auch definiert hatten. Fast gleichzeitig haben drei andere Wissenschaftler eine andere Serie von Codes beschrieben, und zwar unabhängig voneinander einerseits 1959 der französische Mathematiker **Alexis Hocquenghem** (1908–1990) und andererseits 1960 die beiden indischamerikanischen Mathematiker **Raj Chandra Bose** (1901–1987) und **Kumar Ray-Chaudhuri** (geb. 1933). Die von ihnen entdeckten Codes werden nach ihren Initialen **BCH-Codes** genannt. Wir werden BCH-Codes nicht in jedem Detail untersuchen, dennoch folgen in diesem Abschnitt die wichtigsten Ideen. Wir beschränken uns dabei zunächst auf die Beschreibung sog. **primitiver BCH-Codes im engeren Sinne**.

Als wir uns zum ersten Mal mit Reed-Solomon-Codes beschäftigten, hatten wir festgestellt, dass wir zu deren Konstruktion größere Körper benötigten. Daher haben wir insbesondere Körper K mit 2^m Elementen konstruiert, die sich ja für binäre Datenübertragung besonders eignen. Jetzt gehen wir noch einen Schritt weiter und betrachten außer K einen noch größeren Körper, sagen wir L, der K umfasst (d. h. $K \subseteq L$). Beispielsweise könnte $|K| = 2^8$ und $|L| = 2^{16}$ sein. Wie sich gleich herausstellen wird, erleichtert dies die Sache eher, als sie zu verkomplizieren.

Es sei also $|K| = q$ und $|L| = q^s$. Sei weiterhin $n = q^s - 1$. Wie wir wissen, hat auch der Körper L eine primitive n. Einheitswurzel ε, und ε ist damit Nullstelle von $x^n - 1 = (x-1)(x-\varepsilon)\ldots(x-\varepsilon^{n-1})$. Diese Zerlegung von $x^n - 1$ in Faktoren gilt aber nur als Polynom aus $L[x]$, da ε für $s > 1$ nicht in K liegt. Wir können aber auch $x^n - 1$ als Polynom aus $K[x]$ in irreduzible Polynome $f_i(x) \in K[x]$ zerlegen, also $x^n - 1 = f_1(x)\ldots f_r(x)$. Dann muss aber ε Nullstelle von genau einem der $f_i(x)$ sein. Man nennt dann $f_i(x)$ das **Minimalpolynom** von ε über $K[x]$. Ebenso haben die Potenzen ε^j von ε jeweils ein Minimalpolynom.

Mit diesen Vorbetrachtungen können wir BCH-Codes definieren. Seien dazu $1 < d \leq n$ und $g(x)$ das Produkt der verschiedenen Minimalpolynome von $\varepsilon, \varepsilon^2, \ldots, \varepsilon^{d-1}$. Dann heißt der zyklische Code der Länge n über dem Körper K mit Generatorpolynom $g(x)$ **BCH-Code** zur **Auslegungsdistanz** d, und zwar genauer gesagt **primitiv und im engeren Sinne**.

7.2.2 BCH-Schranke

BCH-Codes zur Auslegungsdistanz d sind schon deswegen gut, weil man zeigen kann, dass ihr Minimalabstand stets $\geq d$ ist. Man nennt diese untere Abschätzung auch die **BCH-Schranke**.

Dies ist nicht allzu schwer einzusehen, wenngleich wir dazu wieder die Eigenschaft von Vandermonde-Matrizen verwenden müssen. Sei also $g(x)$ das Generatorpolynom unseres BCH-Codes und $0 \neq c(x) = c_{n-1}x^{n-1} + \ldots + c_0$ ein Codewort. Also lässt sich $c(x)$ schreiben als $c(x) = g(x)f(x)$ mit einem Polynom $f(x)$ und es gilt daher $c(\varepsilon^j) = g(\varepsilon^j)f(\varepsilon^j) = 0$ für $j = 1, \ldots, d-1$. Wir betrachten die Matrix

$$H = \begin{pmatrix} 1 & \varepsilon & \varepsilon^2 & \ldots & \varepsilon^{n-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \varepsilon^{d-1} & (\varepsilon^2)^{d-1} & \ldots & (\varepsilon^{n-1})^{d-1} \end{pmatrix}.$$

Hierbei sind die Zeilen wieder fortlaufende Potenzen der 1. Zeile und Gleiches gilt dann auch für jede quadratische Teilmatrix von H mit $d-1$ Zeilen und Spalten. Dabei handelt es sich wieder um **Vandermonde-Matrizen**, von denen man ja weiß, dass ihre Zeilen- und Spaltenvektoren linear unabhängig sind.

Seien also $s_0, \ldots, s_{n-1}$ die Spaltenvektoren von H. Aus $0 = c(\varepsilon^j) = c_{n-1}(\varepsilon^j)^{n-1} + c_1(\varepsilon^j) + c_0$ für $j = 1, \ldots, d-1$ folgt dann $0 = c_{n-1}s_{n-1} + \ldots + c_1s_1 + c_0s_0$ und somit müssen mindestens d der $c_i \neq 0$ sein. Sonst nämlich wären $d-1$ Spaltenvektoren s_i von H linear abhängig. Damit ist der Minimalabstand des BCH-Codes mindestens d.

BCH-Codes hatten allerdings zur Zeit ihrer Entdeckung den gleichen Nachteil wie Reed-Solomon-Codes: Es gab keinen effizienten Decodieralgorithmus.

7.2.3 Beispiel: BCH-Codes

Reed-Solomon-Codes als BCH-Codes

Wir kommen zunächst zur Einordnung der Reed-Solomon-Codes in die Klasse der BCH-Codes: Reed-Solomon-Codes sind nämlich BCH-Codes – und zwar primitiv und im engeren Sinne.

Das ist bei unserem jetzigen Kenntnisstand völlig klar. Wir betrachten für BCH-Codes nämlich einfach nur den Spezialfall $L = K$. Dann ist $n = q-1$ und $\varepsilon = \beta$ ist eine primitive n. Einheitswurzel in K. Die Minimalpolynome von $\varepsilon^i = \beta^i$ sind die Polynome $x - \beta^i \in K[x]$ und folglich ist das Produkt $g(x) = (x-\beta)(x-\beta^2)\ldots(x-\beta^{d-1})$ das Generatorpolynom unseres BCH-Codes. Dabei handelt es sich aber um genau das Generatorpolynom des Reed-Solomon-Codes $RS_q(d)$, das wir im letzten Abschnitt konstruiert haben.

Wir werden in den folgenden Beispielen $K = \mathbb{Z}_2$, $|L| = 2^s$, $n = 2^s - 1$ und in L eine primitive n. Einheitswurzeln wählen. Daher geben wir die vorübergehende – der Unterscheidung dienende – Bezeichnung ε wieder auf und verwenden hierfür wieder standardmäßig β.

Binäre BCH-Codes der Länge 7

Zunächst ein wirklich einfaches Beispiel, an dem man aber das Konstruktionsprinzip ganz gut verstehen kann. Wir nehmen dazu die Körper $K = \mathbb{Z}_2$ und L mit $|L| = 2^3 = 8$. Dann ist $n = 8 - 1 = 7$ und $x^n - 1 = x^7 - 1 = x^7 + 1$. Offenbar gilt $x^7 + 1 = f_1(x) f_2(x) f_3(x) = (x + 1)(x^3 + x + 1)(x^3 + x^2 + 1)$ mit irreduziblen Polynomen $f_i(x) \in \mathbb{Z}_2[x]$. Für unsere primitive 7. Einheitswurzel $\beta = t$ aus dem letzten Abschnitt gilt $f_2(\beta) = 0$, aber auch $f_2(\beta^2) = f_2(\beta^4) = 0$. Andererseits ist $f_3(\beta^3) = f_3(\beta^5) = f_3(\beta^6) = 0$. Somit ist $f_2(x)$ Minimalpolynom von β, β^2, β^4 und $f_3(x)$ Minimalpolynom von β^3, β^5, β^6. Der BCH-Code der Länge $n = 7$ über $\mathbb{Z}_2$ zur Auslegungsdistanz $d = 3$ hat also als Generatorpolynom $g(x) = f_2(x)$, der zur Auslegungsdistanz $d = 4$ das Generatorpolynom $g(x) = f_2(x) f_3(x)$. Für $d = 3$ ergibt sich also die Dimension $k = n - \text{grad}\,(g(x)) = 7 - 3 = 4$, für $d = 4$ entsprechend $k = 7 - 6 = 1$.

Nochmals zum Vergleich: Hätten wir stattdessen den Reed-Solomon-Code $RS_8(d)$ über dem Körper L mit $2^3 = 8$ Elementen betrachtet, dann wäre für $d = 3$ das Generatorpolynom $g(x) = (x - \beta)(x - \beta^2) \in L[x]$ und für $d = 4$ das Generatorpolynom $g(x) = (x - \beta)(x - \beta^2)(x - \beta^3) \in L[x]$ gewesen.

Binärer BCH-Code der Länge 15

Ein etwas größeres Beispiel erhält man für $K = \mathbb{Z}_2$, $|L| = 2^4 = 16$ und $n = 16-1 = 15$. In diesem Fall ist $x^{15} + 1 = f_1(x) f_2(x) f_3(x) f_4(x) f_5(x) = (x + 1)(x^2 + x + 1)(x^4 + x + 1)(x^4 + x^3 + 1)(x^4 + x^3 + x^2 + x + 1)$ mit irreduziblen Polynomen $f_i(x) \in \mathbb{Z}_2[x]$, wie es nachzurechnen gilt. Außerdem kann man für unsere primitive 15. Einheitswurzel $\beta = t$ aus dem letzten Abschnitt nachprüfen, dass $f_3(\beta) = f_3(\beta^2) = f_3(\beta^4) = 0$, $f_5(\beta^3) = f_5(\beta^6) = 0$ und $f_2(\beta^5) = 0$ ist. Betrachtet man nun den binären zyklischen Code der Länge $n = 15$ mit dem Generatorpolynom $g(x) = f_2(x) f_3(x) f_5(x) = (x^2 + x + 1)(x^4 + x + 1)(x^4 + x^3 + x^2 + x + 1) = x^{10} + x^8 + x^5 + x^4 + x^2 + x + 1$, dann ist dies der BCH-Code zur Auslegungsdistanz $d = 7$. Er hat Dimension $k = n - \text{grad}\,(g(x)) = 15 - 10 = 5$ und Minimalabstand $d = 7$, kann also sicher bis zu drei Fehler korrigieren. Dies ist auch gleichzeitig ein konkretes Praxisbeispiel, wie wir im nächsten Abschnitt über QR-Barcodes lernen werden.

7.2.4 Güte von BCH-Codes

Für die Güte von BCH-Codes gilt grundsätzlich Ähnliches wie für Reed-Solomon-Codes. Ein Nachteil von BCH-Codes gegenüber Reed-Solomon-Codes ist, dass mit Vorgabe der Auslegungsdistanz d der Minimalabstand nur abgeschätzt werden kann, immerhin aber

Tab. 7.1 Parameter kleiner nichttrivialer binärer BCH-Codes

n	k	e
7	4	1
15	11	1
	7	2
	5	3
31	26	1
	21	2
	16	3
	11	5
	6	7
63	57	1
	51	2
	45	3
	39	4
	36	5
	30	6
	24	7
	18	10
	16	11
	10	13
	7	15

nach unten. Eines aber sollte aus den obigen Beispielen klar geworden sein: Ein Vorteil von BCH-Codes ist, dass man oft wieder mit binären Codes arbeiten kann. Dies wird auch gleich nochmals bei einer anderen Praxisanwendung deutlich. In Tab. 7.1 aber zuerst zur Illustration die Parameter kleiner nichttrivialer binärer BCH-Codes der Länge n, der Dimension k und mit Fehlerkorrekturkapazität e, folglich mit Minimalabstand $\geq 2e+1$.

7.2.5 Anwendung: Pagen mit BCH-Codes

Mit bei Behörden (Feuerwehr, Krankenhäuser etc.) eingesetzten Personenrufempfängern (**Pager**) können Textnachrichten auf einem kleinen Display angezeigt werden. Der Empfänger erhält dabei Datenpakete mit 32 Bits, wobei 21 Bits zur Informationsübermittlung dienen und 11 redundante Bits für die automatische Fehlerkorrektur sorgen. Und so funktioniert es. Die Codierung erfolgt zunächst mit einem binären BCH-Code C der Länge 31, d. h. $K = \mathbb{Z}_2$ und $|L| = 2^5 = 32$. Genutzt wird dazu das Generatorpolynom $g(x) = x^{10}+x^9+x^8+x^6+x^5+x^3+1 = (x^5+x^2+1)(x^5+x^4+x^3+x^2+1) = f_1(x)\,f_2(x)$ mit irreduziblen Polynomen $f_i(x)$, die ihrerseits Teiler von $x^{31}+1$ sind. Der BCH-Code hat somit Dimension $k = n - \operatorname{grad}(g(x)) = 31-10 = 21$. Codiert wird in systemati-

scher Form, d. h. 21 Informationsbits gefolgt von 10 Redundanzbits. Anschließend wird zum Code C noch der erweiterte Code $C^{\wedge}$ gebildet, d. h., es wird ein Paritätsprüfungsbit ergänzt, das sicherstellt, dass die Summe der Einträge gleich 0 ist. Damit ergibt sich also insgesamt ein $[32, 21]$-Code. Wie man nachrechnen kann, ist für die primitive 31. Einheitswurzel $\beta = t$ aus dem letzten Abschnitt $f_1(\beta) = f_1(\beta^2) = f_1(\beta^4) = 0$ und $f_2(\beta^3) = 0$ und damit ist C der BCH-Code zur Auslegungsdistanz $d = 5$. Folglich ist sein Minimalabstand mindestens 5, wie auch aus Tab. 7.1 ersichtlich ist. Als Erweiterung eines binären Codes hat $C^{\wedge}$ gerades Minimalgewicht und daher Minimalabstand mindestens 6. Damit können zwei Fehler sicher korrigiert und drei Fehler sicher erkannt werden.

7.2.6 Verallgemeinerte BCH-Codes

Wir wollen abschließend noch kurz erwähnen, was es in der Definition von BCH-Codes mit *primitiv im engeren Sinne* auf sich hat.

- Gibt es einen echten Teil m von n, so kann man aus einer primitiven n. Einheitswurzel ε auch eine primitive m. Einheitswurzel machen durch $\eta = \varepsilon^{(n/m)}$. Auf diese Weise kann man mittels $\eta, \eta^2, \ldots, \eta^{d-1}$ BCH-Codes der Länge m definieren, die dann *nicht mehr primitiv* heißen.
- Wählt man zusätzlich eine natürliche Zahl $r > 1$ und konstruiert zu ε^r, ε^{r+1}, …, ε^{r+d-2} den entsprechenden BCH-Code, dann heißt dieser *nicht mehr im engeren Sinne*.

Aber auch für solche verallgemeinerte BCH-Codes gilt die BCH-Schranke, d. h., der Minimalabstand ist größer oder gleich d. Die letzten beiden Abschnitte waren sicher ein wenig anstrengend. Wir können uns davon im nächsten Abschnitt etwas erholen, indem wir ausführlich eine Anwendung der Reed-Solomon- und BCH-Codes besprechen: die QR-Codes.

7.3 QR-Code – Quick Response

Wir haben 2-D-Barcodes schon in Abschn. 5.3 angesprochen. Hier wollen wir uns konkreter mit dem QR-Code beschäftigen. Der **QR-Code** (**Quick Response**, dt. schnelle Antwort) wurde von der japanischen Firma Denso Wave im Jahr 1994 entwickelt. Die Spezifikation wurde von Denso Wave offengelegt, die Verwendung ist lizenz- und kostenfrei. Der QR-Code enthält neben der eigentlichen Information redundante Daten für ein automatisches Fehlerkorrekturverfahren. Für Smartphones gibt es viele Apps, die den QR-Code lesen und interpretieren können. Dabei werden vom Code nur Steuerungsdaten bereitgestellt (z. B. URLs). Die eigentliche Funktion liefert die App.

7.3.1 Genereller Aufbau des QR-Codes

Bevor wir uns etwas näher mit dem Aufbau des **QR-Codes** beschäftigen, zunächst das Wichtigste vorweg. QR-Matrizen werden in vier verschiedenen sog. **Fehlerkorrekturstufen** angeboten, nach dem Motto: Je sensibler die Anwendung ist, für die der jeweilige QR-Code eingesetzt wird, umso höher sollte die Fehlerkorrekturstufe sein. Angeboten werden diese vier Stufen.

- L Fehlerkorrekturstufe 7 %
- M Fehlerkorrekturstufe 15 %
- Q Fehlerkorrekturstufe 25 %
- H Fehlerkorrekturstufe 30 %

Dies ist wie folgt zu verstehen: Hat der QR-Code eine Fehlerkorrekturstufe von x %, so können bis zu x % der QR-Matrix beschädigt oder verdeckt sein (z. B. beschmiert) und es ist trotzdem möglich, die im Code enthaltene Information korrekt auszulesen. Wir werden uns also dabei wieder um die Korrektur von Fehlerbündeln kümmern müssen. Allerdings können wir nicht unbedingt von Auslöschungen ausgehen, da etwa bei Beschmutzung der Scanner die entsprechenden Stellen doch als schwarz oder weiß erkennt und einliest. Abb. 7.2 zeigt einen typischen QR-Code. Charakteristisch sind die *Bullaugen* rechts und links oben sowie links unten.

QR-Codes sind wie jeder 2-D-Barcode in einer Schachbrettform (Matrix) aufgebaut. Jedes einzelne Feld – auch **Matrixzelle** oder **Modul** genannt – beinhaltet die Information eines Bits:

- schwarz $= 1$,
- weiß $= 0$.

Eine QR-Matrix hat mindestens 21 und maximal 177 Zeilen und Spalten, also mindestens 21×21 und maximal 177×177 Matrixzellen. Wir betrachten hier stets den sog. Standard-QR-Code (also keine Micro-QR-Codes).

Die Information eines QR-Codes kann in verschiedenen **Modi** abgebildet sein: numerisch, alphanumerisch, Bytes und Kanji (japanische Zeichen). Jedem String von **Datenmodulen** wird deshalb ein 4-Bits-String zur Festlegung des Modus vorangestellt. Dies ermöglicht es, in einem QR-Code mehrere Modi zu verwenden. Beim gebräuchlichen

Abb. 7.2 Beispiel eines QR-Codes (Quelle [WVQR1])

Bytemodus des QR-Codes werden immer acht angrenzende Matrixzellen zu einem Byte zusammengefasst, mit dem sich dann 256 Zeichen codieren lassen. Diese Datenbytes der QR-Matrix werden von der unteren rechten Matrixecke ausgehend in die zwei rechten Spalten aufwärts in einer Art *Zickzackmuster* geschrieben. Oben angekommen, geht es die nächsten beiden Spalten entsprechend wieder abwärts usw. Die Mustermodule, die wir jetzt beschreiben, werden bei dieser Prozedur einfach übersprungen.

Man muss nämlich beachten, dass normalerweise Scanner den Code nie im 90°-Winkel einlesen. Sobald eine Bildaufnahme in einem anderen Winkel erfolgt, erscheint der Code im Bild verzerrt. Die Decodiersoftware muss diese Verzerrung mithilfe der Muster erkennen und kompensieren. Hier sind die verwendeten **Mustermodule**, die auch in Abb. 7.3 visualisiert sind:

- drei markante *Bullaugen* zur Lagebestimmung in den Ecken,
- hell-dunkel-alternierende *Taktmodule* zur Gittererkennung, als Verbindungslinie zwischen den *Bullaugen* in x- und y-Richtung,
- *Kleinere in der Matrix verteilte Bullaugen* zur Vermeidung von Verzerrungen (nur ab Matrixgröße 25×25).

Außerdem gibt es noch **Format-** und **Versionsmodule**.

- Die **Formatmodule** – in Abb. 7.3 an allen drei Ecken – legen die Fehlerkorrekturstufe und das Muster zur Maskierung fest.
- Die **Versionsmodule** – in Abb. 7.3 an den Ecken links unten und rechts oben – beinhalten die Versionsnummer, die wiederum die Matrixgröße festlegt.

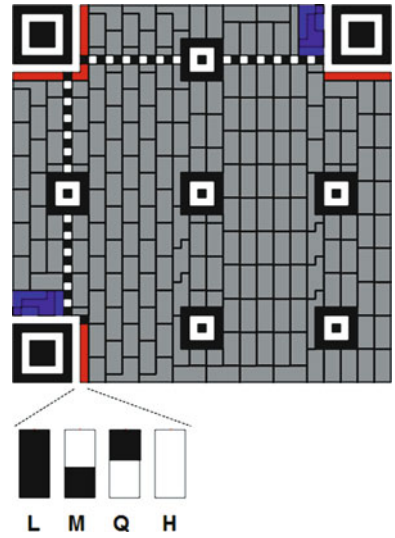


Abb. 7.3 Muster-, Format- und Versionsmodule beim QR-Code (Quelle [WVQR2]) sowie die Fehlerkorrekturstufen

Abb. 7.4 Maskierung beim QR-Code (Quelle [WVQR1])

Maskierung wird eingesetzt, um Matrixdesigns zu vermeiden, die den Scanner *irritieren* könnten (z. B. Bereiche, die den Mustermodulen ähneln, oder große weiße Flächen). Konkret werden bei der Maskierung bestimmte Module invertiert (d. h. aus schwarz wird weiß und umgekehrt), während die anderen unverändert bleiben. Beim Codedesign werden daher die acht vordefinierten Muster ausprobiert und das beste dem QR-Code als Maskierungsinformation in den Formatmodulen beigefügt. Bei der Anwendung erkennt der Scanner dieses Maskenformat und verwendet es automatisch beim Auslesen der QR-Matrix. Abb. 7.4 zeigt das Prinzip:

- links eine QR-Matrix, wie sie etwa in einer Zeitung abgedruckt zu sehen ist,
- in der Mitte die zugehörige *Maske*, bei der die nicht zu den Datenblöcken gehörenden Lage- und Taktmodulen entsprechend ausgespart sind,
- rechts das QR-Bild, wie es der Scanner dann umsetzt, nämlich mit einer XOR-Verknüpfung der Bits, d. h. mit Addition modulo 2.

7.3.2 Anwendung: Codierung der Datenbytes im QR-Code

Wie oben schon angedeutet, werden die Größe der QR-Matrix und damit auch die Aufnahmekapazität an Bytes durch die Versionsnummer festgelegt, was allerdings im Zusammenhang mit dem Begriff Version eher verwirrend erscheint. Wir wollen hier zunächst auf die Bytes der Datenmodule eingehen. Einige dieser Datenbytes tragen *echte* Information, andere sind redundant und dienen der Fehlerkorrektur, wobei das Verhältnis der beiden von der Fehlerkorrekturstufe des QR-Codes abhängt. Beim QR-Code werden hierzu, abhängig von der Versionsnummer und der Fehlerkorrekturstufe, verkürzte Reed-Solomon-Codes $RS_q(d)$ verwendet, die ja wiederum MDS-Codes sind (siehe Abschn. 5.4). Hier zunächst einige generelle Bemerkungen.

- Wie bei bytebasierten Strukturen üblich, verwendet man den Körper K mit $q = 2^8 = 256$ Elementen. Daher ist die auf n verkürzte Blocklänge durch 255 begrenzt.
- Die Codierung erfolgt in systematischer Form, d. h. jeder Block hat k Informationsbytes, gefolgt von $n - k$ Paritätsbytes.
- Die Gesamtkapazität an Datenbytes wird dabei in ein oder mehrere Blöcke von verkürzten Reed-Solomon-Codes mit unterschiedlicher Länge aufgeteilt, bei denen jeder

Block höchstens $n - k \leq 30$ Paritätsbytes hat, wegen $d = n - k + 1 \leq 31$ also höchstens 15-fehlerkorrigierend ist.
- Die Anzahl der Informationsbytes wird bei gleicher Versionsnummer von einer Fehlerkorrekturstufe zur nächsten immer geringer. Daher muss man, um die gleiche Information abzubilden, möglicherweise auf höherer Versionen ausweichen.
- Zusätzlich wird bei QR-Codes das Mittel des Block-Interleavings eingesetzt, um Fehlerbündel zu korrigieren. Dazu werden die jeweiligen Reed-Solomon-Codewörter zeilenweise in eine temporäre Matrix geschrieben, spaltenweise ausgelesen und so als 2-D-Barcode generiert und ausgedruckt.
- Die Decodierung ist bei solch kleinen Codes *kein* Problem. Zuerst wird das Interleaving wieder rückgängig gemacht, d. h., der gescannte Code wird spaltenweise in eine Matrix geschrieben und zeilenweise ausgelesen. Danach wird der Code mittels Syndrom decodiert.

Wir betrachten nun konkret das Beispiel Version 5 mit einer Gesamtkapazität von 134 Datenbytes. Hier sind die Parameter der verkürzten Reed-Solomon-Codes.

- Stufe L $[134, 108, 27]$
- Stufe M $2 \times [67, 43, 25]$ und Interleaving-Tiefe 2
- Stufe Q $2 \times [33, 15, 19]$ und $2 \times [34, 16, 19]$ und Interleaving-Tiefe 4
- Stufe H $2 \times [33, 11, 23]$ und $2 \times [34, 12, 23]$ und Interleaving-Tiefe 4

Wir wollen uns das Fehlerkorrekturverhalten im Einzelnen anschauen.

Bei Stufe L ist es einfach, der Code kann wegen $d = 27$ bis zu 13 Fehler korrigieren, dies sind ca. 10 % von 134 Bytes, also mehr als die versprochenen 7 %.

Bei Stufe M wird jetzt zusätzlich das Interleaving wirksam. Denn die beiden Codes werden ja in die zwei Zeilen einer temporären Matrix geschrieben, spaltenweise ausgelesen und so als QR-Matrix ausgedruckt. Man geht nun davon aus, dass es sich bei den Fehlabtastungen um kleinere oder größere Fehlerbündel handelt, die dann einige Spalten der temporären Matrix betreffen und sich so im Mittel auf die beiden einzelnen Reed-Solomon-Codes aufteilen. Wegen $d = 25$ können die beiden Codes jeweils bis zu zwölf Fehler korrigieren, also insgesamt 24. Das sind ca. 18 %, also mehr als die versprochenen 15 %. Wäre die Aufteilung der Fehler also sehr asymmetrisch bezüglich beider Reed-Solomon-Codes – was nicht realitätsnah ist –, so könnte man die 15 % ggf. nicht mehr ganz einhalten.

Bei Stufe Q und H argumentiert man analog, nur mit einer temporären Matrix mit vier statt mit zwei Zeilen. Für Stufe Q sind alle vier Codes 9-fehlerkorrigierend, per Interleaving können also bis zu 36 Fehler korrigiert werden. Dies sind ca. 27 % und damit mehr als 25 %. Bei Stufe H schließlich handelt es sich um vier 11-fehlerkorrigierende Codes, also können insgesamt 44 und damit 33 % der Fehler korrigiert werden. Also sind auch hier die 30 % eingehalten.

Tab. 7.2 Maskierungsmuster innerhalb der Formatbits des QR-Codes (Quelle [WVQR2])

Code	Maskierung
000	$(i + j)\,\%2 = 0$
001	$i\,\%2 = 0$
010	$j\,\%3 = 0$
011	$(i + j)\,\%3 = 0$
100	$((i/2) + (j/3))\,\%2 = 0$
101	$(i * j)\,\%2 = 0$ und $(i * j)\,\%3 = 0$
110	$(i * j + (i * j)\,\%3)\,\%2 = 0$
111	$(i + j + (i * j)\,\%3)\,\%2 = 0$

Wer Spaß hat, kann das noch für andere Versionen kontrollieren, die gesamte Tabelle der Reed-Solomon-Codes findet man am Ende des Abschnitts.

7.3.3 Anwendung: Codierung der Format- und Versionsbits im QR-Code

Die **Formatbits** sind 5-stellig und bestehen aus zwei Bits für die Fehlerkorrekturstufe und drei Bits für die verwendete Maskierung. Dabei sind die Fehlerkorrekturstufen wie folgt codiert: L $= 01$, M $= 00$, Q $= 11$ und H $= 10$. Die zulässigen Maskierungsmuster zur Invertierung von Modulen ist in Tab. 7.2 wiedergegeben. Hierbei bezeichnen i und j die Zeile bzw. Spalte des jeweiligen Moduls und % die Addition modulo 2 bzw. 3.

Die fünf Formatbits werden in eine 15-stellige Bitfolge codiert, und zwar als BCH-Code. Genauer gesagt handelt es sich um den BCH-Code für $K = \mathbb{Z}_2$, $|L| = 2^4 = 16$ und $n = 15$, den wir im letzten Abschnitt vorgestellt haben, mit Generatorpolynom $g(x) = (x^2 + x + 1)(x^4 + x + 1)(x^4 + x^3 + x^2 + x + 1) = x^{10} + x^8 + x^5 + x^4 + x^2 + x + 1$, Dimension $n - \text{grad}\,(g(x)) = 15 - 10 = 5$ und Minimalabstand 7; er kann also bis zu drei Fehler korrigieren. Die Codierung erfolgt wieder in systematischer Form. Hier noch ein konkretes Beispiel. Wir wollen die Fehlerkorrekturstufe M ($= 00$) und die Maske = 001 verwenden. Die BCH-Codierung liefert dann das Codewort 000010100110111. Aber auch beim Format erfolgt eine Maskierung, und zwar stets mit dem fest vorgegebenen Wort 101010000010010. Dieses wird mit einer XOR-Verknüpfung der Bits, d. h. mit Addition modulo 2, dem BCH-Codewort *übergestülpt*. Es ergibt sich demnach 101000100100101 und dieser String wird in Form von Modulen in der QR-Matrix angedruckt. Da es sich nur um insgesamt 32 Codewörter handelt, ist bei der Decodierung der Formatbits die vollständige Suche, also Hamming-Decodierung, am effektivsten.

Außer mit dem BCH-Code sind die Formatbits noch zusätzlich geschützt. Sie stehen nämlich redundant um das linke obere *Bullauge* als auch in zwei Teile gebrochen um die beiden anderen *Bullaugen*. Dies entspricht also einem zusätzlichen Wiederholungscode.

Die **Versionsbits** sind ebenfalls redundant abgelegt, weit voneinander entfernt am linken unteren und rechten oberen *Bullauge*. Es handelt sich also hierbei ebenfalls um einen

Wiederholungscode. Als weiterer Quercheck für die Versionsnummer dient die Zeilen- und Spaltenanzahl der Matrix.

7.3.4 Anwendung: Liste der Reed-Solomon-Codes für Datenbytes im QR-Code

In Tab. 7.3 findet man zusammengestellt:

- Versionsnummer,
- Zeilen-/Spaltenanzahl,
- Kapazität an Datenbytes,
- Parameter $[n, k]$ der verkürzten Reed-Solomon-Codes für alle Fehlerkorrekturstufen L, M, Q und H.

7.4 PGZ-Decodierung

Wir kommen nun zurück zu der Frage, wie man Reed-Solomon-Codes und BCH-Codes effizient und schnell decodieren kann. Wie bereits erwähnt, gab es bei deren Entdeckung nur Verfahren, die bei größeren Codes an ihre Grenzen stießen. Bereits kurz nach der Publikation der BCH- und Reed-Solomon-Codes wurde im Jahr 1960 von **William Westley Peterson** ein Decodieralgorithmus angegeben, der 1961 von **Daniel Gorenstein** und **Neal Zierler** auf nichtbinäre Codes erweitert wurde. **Peterson** kennen wir schon von den Cyclic Redundancy Checks (CRC). **Daniel Gorenstein** (1923–1992) war ein amerikanischer Wissenschaftler, der eher *zufällig* auch in der Codierungstheorie einen Beitrag geleistet hat. Weltbekannt wurde er aber im Zusammenhang mit dem *Klassifikationssatz endlicher einfacher Gruppen*, einem Ergebnis einer großen Gruppe internationaler Wissenschaftler, das wir auch schon im Umfeld der Golay-Codes kurz erwähnt hatten. **Neal Zierler** hingegen war ein amerikanischer Mathematiker, der hauptsächlich im Bereich Codierungstheorie arbeitete. Der Algorithmus, den wir nun besprechen, ist unter der Bezeichnung **PGZ-Decodierung** bekannt.

7.4.1 PGZ-Decodierung – Warum der Algorithmus funktioniert

Wie bei Decodierverfahren ja meist üblich, wird es nun wieder kurz etwas kniffelig. Wir bemühen uns aber, den wesentlichen Kern herauszuarbeiten und das im Anschluss auch wieder mit kleinen Beispielen zu illustrieren.

Wir sammeln zunächst wieder einige Bezeichnungen, beschränken uns bei unseren Überlegungen aber auf Reed-Solomon-Codes. Sei also $C = RS_q(d)$, $|K| = q$, $n = q-1$,

$\dim(C) = k = n - d + 1$ und β eine primitive n. Einheitswurzel in K. Wir haben uns in Abschn. 7.1 auch eine Kontrollmatrix von C überlegt, nämlich

$$H = \begin{pmatrix} 1 & \beta & \beta^2 & \dots & \beta^{n-1} \\ 1 & \beta^2 & (\beta^2)^2 & \dots & (\beta^{n-1})^2 \\ 1 & \beta^3 & (\beta^2)^3 & \dots & (\beta^{n-1})^3 \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & \beta^{d-1} & (\beta^2)^{d-1} & \dots & (\beta^{n-1})^{d-1} \end{pmatrix}.$$

Sei also $c = (c_0, \dots, c_{n-1}) \in C$ ein Codewort und $v = c + f$ das empfangene Wort mit Fehlervektor $f = (f_0, \dots, f_{n-1}) \in K^n$. Sei e die Fehlerkorrekturkapazität von C, also e maximal mit $2e + 1 \leq d$. Wir nehmen weiter an, dass höchstens e Fehler aufgetreten sind, d. h. $t = wt(f) \leq e$. Es sei außerdem $(s_1, \dots, s_{d-1})$ das Syndrom von v und f, das sich mit den Zeilenvektoren z_j von H berechnet als $s_j = \langle v, z_j \rangle = \langle f, z_j \rangle$. Ferner betrachten wir auch wieder den Träger $Tr(f) = \{m_1, \dots, m_t\}$ von f, d. h. die Positionen m von f, an denen $f_m \neq 0$ ist.

Zusätzlich – und das ist *neu* – definieren wir das **Fehlerortungspolynom** als $q(x) = \prod_m (1 - \beta^m x) = q_0 + q_1 x + \dots + q_t x^t \in K[x]$, wobei das Produkt über $m \in Tr(f)$ läuft. Dabei ist $t \leq e$, $q_0 = 1$ und $m \in Tr(f)$ genau dann, wenn $q(\beta^{-m}) = q(\beta^{n-m}) = 0$. Damit steht auch bereits unsere *Globalstrategie* bei der PGZ-Decodierung.

- Bestimme $q(x)$ und damit $Tr(f)$.
- Bestimme mittels $Tr(f)$ den Fehlervektor f.

Wir verschaffen uns dazu aber zuerst zwei Gleichungssysteme (7.1) und (7.2), die – bei richtiger Interpretation – der Schlüssel für die PGZ-Decodierung sein werden.

Für $i = 1, \dots, d-1$ gilt einerseits

$$s_i = \langle f, z_i \rangle = 1 f_0 + \beta^i f_1 + \beta^{2i} f_2 + \dots + \beta^{(n-1)i} f_{n-1} = \sum_m \beta^{mi} f_m, \tag{7.1}$$

wobei die Summe über $m \in Tr(f)$ läuft.

Daraus leiten wir für $i = t+1, \dots, d-1$ die Aussage $q_0 s_i + q_1 s_{i-1} + \dots + q_t s_{i-t} = q_0 \sum_m \beta^{mi} f_m + q_1 \sum_m \beta^{m(i-1)} f_m + \dots + q_t \sum_m \beta^{m(i-t)} f_m = \sum_m \beta^{mi} f_m (q_0 + q_1 \beta^{-m} + \dots + q_t \beta^{-mt}) = \sum_m \beta^{mi} f_m q(\beta^{-m})$ ab. Dies wiederum ist gleich 0, da die Summe über $m \in Tr(f)$ läuft. Wegen $q_0 = 1$ folgt daraus also anderseits, dass

für $i = t+1, \dots, d-1$ die Gleichung $-s_i = q_1 s_{i-1} + \dots + q_t s_{i-t}$ gilt. (7.2)

Jetzt müssen wir uns noch überlegen, wie wir (7.1) und (7.2) für unsere *Globalstrategie* verwenden können.

Tab. 7.3 Reed-Solomon-Codierungen der Datenbytes für sämtliche Versionen des QR-Codes (Quelle [WVQR2])

Versions-nummer	Zeilen/ Spalten-anzahl	Kapazität Daten-bytes	RS-Code $[n, k]$ Stufe L	RS-Code $[n, k]$ Stufe M	RS-Code $[n, k]$ Stufe Q	RS-Code $[n, k]$ Stufe H
1	21	26	(26, 19)	(26, 16)	(26, 13)	(26, 9)
2	25	44	(44, 34)	(44, 28)	(44, 22)	(44, 16)
3	29	70	(70, 55)	(70, 44)	2×(35, 17)	2×(35, 13)
4	33	100	(100, 8)	2×(50, 32)	2×(50, 24)	4×(25, 9)
5	37	134	(134, 108)	2×(67, 43)	2×(33, 15), 2×(34, 16)	2×(33, 11), 2×(34, 12)
6	41	172	2×(86, 68)	4×(43, 27)	4×(43, 19)	4×(43, 15)
7	45	196	2×(98, 78)	4×(49, 31)	2×(32, 14), 4×(33, 15)	4×(39, 13), (40, 14)
8	49	242	2×(121, 97)	2×(60, 38), 2×(61, 39)	4×(40, 18), 2×(41, 19)	4×(40, 14), 2×(41, 15)
9	53	292	2×(146, 116)	3×(58, 36), 2×(59, 37)	4×(36, 16), 4×(37, 17)	4×(36, 12), 4×(37, 13)
10	57	346	2×(86, 68), 2×(87, 69)	4×(69, 43), (70, 44)	6×(43, 19), 2×(44, 20)	6×(43, 15), 2×(44, 16)
11	61	404	4×(101, 81)	(80, 50), 4×(81, 51)	4×(50, 22), 4×(51, 23)	3×(36, 12), 8×(37, 13)
12	65	466	2×(116, 92), 2×(117, 93)	6×(58, 36), 2×(59, 37)	4×(46, 20), 6×(47, 21)	7×(42, 14), 4×(43, 15)
13	69	532	4×(133, 107)	8×(59, 37), (60, 38)	8×(44, 20), 4×(45, 21)	12×(33, 11), 4×(34, 12)
14	73	581	3×(145, 115), (146, 116)	4×(64, 40), 5×(65, 41)	11×(36, 16), 5×(37, 17)	11×(36, 12), 5×(37, 13)
15	77	655	5×(109, 87), (110, 88)	5×(65, 41), 5×(66, 42)	5×(54, 24), 7×(55, 25)	11×(36, 12), 7×(37, 13)
16	81	733	5×(122, 98), (123, 99)	7×(73, 45), 3×(74, 46)	15×(43, 19), 2×(44, 20)	3×(45, 15), 13×(46, 16)
17	85	815	(135, 107), 5×(136, 108)	10×(74, 46), (75, 47)	(50, 22), 15×(51, 23)	2×(42, 14), 17×(43, 15)
18	89	901	5×(150, 120), (151, 121)	9×(69, 43), 4×(70, 44)	17×(50, 22), (51, 23)	2×(42, 14), 19×(43, 15)
19	93	991	3×(141, 113), 4×(142, 114)	3×(70, 44), 11×(71, 45)	17×(47, 21), 4×(48, 22)	9×(39, 13), 16×(40, 14)
20	97	1085	3×(135, 107), 5×(136, 108)	3×(67, 41), 13×(68, 42)	15×(54, 24), 5×(55, 25)	15×(43, 15), 10×(44, 16)

Tab. 7.3 (Fortsetzung)

Versions-nummer	Zeilen/ Spalten-anzahl	Kapazität Daten-bytes	RS-Code $[n, k]$ Stufe L	RS-Code $[n, k]$ Stufe M	RS-Code $[n, k]$ Stufe Q	RS-Code $[n, k]$ Stufe H
21	101	1156	4×(144, 116), 4×(145, 117)	17×(68, 42)	17×(50, 22), 6×(51, 23)	19×(46, 16), 6×(47, 17)
22	105	1258	2×(139, 111), 7×(140, 112)	17×(74, 46)	7×(54, 24), 16×(55, 25)	34×(37, 13)
23	109	1364	4×(151, 121), 5×(152, 122)	4×(75, 47), 14×(76, 48)	11×(54, 24), 14×(55, 25)	16×(45, 15), 14×(46, 16)
24	113	1474	6×(147, 117), 4×(148, 118)	6×(73, 45), 14×(74, 46)	11×(54, 24), 16×(55, 25)	30×(46, 16), 2×(47, 17)
25	117	1588	8×(132, 106), 4×(133, 107)	8×(75, 47), 13×(76, 48)	7×(54, 24), 22×(55, 25)	22×(45, 15), 13×(46, 16)
26	121	1706	10×(142, 114), 2×(143, 115)	19×(74, 46), 4×(75, 47)	28×(50, 22), 6×(51, 23)	33×(46, 16), 4×(47, 17)
27	125	1828	8×(152, 122), 4×(153, 123)	22×(73, 45), 3×(74, 46)	8×(53, 23), 26×(54, 24)	12×(45, 15), 28×(46, 16)
28	129	1921	3×(147, 117), 10×(148, 118)	3×(73, 45), 23×(74, 46)	4×(54, 24), 31×(55, 25)	11×(45, 15), 31×(46, 16)
29	133	2051	7×(146, 116), 7×(147, 117)	21×(73, 45), 7×(74, 46)	(53, 23), 37×(54, 24)	19×(45, 15), 26×(46, 16)
30	137	2185	5×(145, 115), 10×(146, 116)	19×(75, 47), 10×(76, 48)	15×(54, 24), 25×(55, 25)	23×(45, 15), 25×(46, 16)
31	141	2323	13×(145, 115), 3×(146, 116)	2×(74, 46), 29×(75, 47)	42×(54, 24), (55, 25)	23×(45, 15), 28×(46, 16)
32	145	2465	17×(145, 115)	10×(74, 46), 23×(75, 47)	10×(54, 24), 35×(55, 25)	19×(45, 15), 35×(46, 16)
33	149	2611	17×(145, 115), (146, 116)	14×(74, 46), 21×(75, 47)	29×(54, 24), 19×(55, 25)	11×(45, 15), 46×(46, 16)
34	153	2761	13×(145, 115), 6×(146, 116)	14×(74, 46), 23×(75, 47)	44×(54, 24), 7×(55, 25)	59×(46, 16), (47, 17)
35	157	2876	12×(151, 121), 7×(152, 122)	12×(75, 47), 26×(76, 48)	39×(54, 24), 14×(55, 25)	22×(45, 15), 41×(46, 16)
36	161	3034	6×(151, 121), 14×(152, 122)	6×(75, 47), 34×(76, 48)	46×(54, 24), 10×(55, 25)	2×(45, 15), 64×(46, 16)
37	165	3196	17×(152, 122), 4×(153, 123)	29×(74, 46), 14×(75, 47)	49×(54, 24), 10×(55, 25)	24×(45, 15), 46×(46, 16)
38	169	3362	4×(152, 122), 18×(153, 123)	13×(74, 46), 32×(75, 47)	48×(54, 24), 14×(55, 25)	42×(45, 15), 32×(46, 16)
39	173	3532	20×(147, 117), 4×(148, 118)	40×(75, 47), 7×(76, 48)	43×(54, 24), 22×(55, 25)	10×(45, 15), 67×(46, 16)
40	177	3706	19×(148, 118), 6×(149, 119)	18×(75, 47), 31×(76, 48)	34×(54, 24), 34×(55, 25)	20×(45, 15), 61×(46, 16)

Nehmen wir dazu zunächst an, wir würden $Tr(f) = \{m_1, \ldots, m_t\}$ schon kennen. Dann betrachten wir die ersten t Gleichungen in (7.1) mit den t Unbestimmten f_m und den bekannten Koeffizienten β^{mi}, also t Gleichungen mit t Unbestimmten. Die Koeffizienten als Matrix geschrieben ergeben folgendes Schema:

$$\begin{pmatrix} \beta^{m_1} & \ldots & \beta^{m_t} \\ \beta^{2m_1} & & \beta^{2m_t} \\ \vdots & & \vdots \\ \beta^{tm_1} & \ldots & \beta^{tm_t} \end{pmatrix},$$

und mit den Spaltenvektoren b_m dieser Matrix liest sich das Gleichungssystem in Vektorschreibweise als $(s_1, \ldots, s_t) = \sum_m f_m b_m$. Nun sollten wir uns erinnern, dass es sich wieder um eine **Vandermonde-Matrix** handelt, mit t Zeilen und t Spalten. Von einer solchen wissen wir aber schon, dass deren t Spaltenvektoren linear unabhängig sind und dass damit dieses Gleichungssystem eine eindeutige Lösung hat. Wir haben somit den Fehlervektor $f = (f_0, \ldots, f_{n-1})$ bestimmt und der zweite Punkt unserer *Globalstrategie* ist abgearbeitet.

Jetzt also zum ersten Punkt. Um $Tr(f)$ zu ermitteln, genügt es, $q(x)$ zu bestimmen. Denn dann ergeben sich die Fehlerpositionen $m \in Tr(f)$ durch Überprüfung, ob $q(\beta^{-m}) = q(\beta^{n-m}) = 0$ gilt.

Machen wir wieder nur einen kleinen Schritt und nehmen an, dass wir zumindest schon die Anzahl $t = |Tr(f)|$ der Fehlerstellen kennen. Wir haben ja vorausgesetzt, dass höchstens e Fehler auftreten, also ist $t \leq e \leq (d-1)/2$. Daher umfasst (7.2) mindestens t Gleichungen. Für beliebige r mit $t \leq r \leq e$ betrachten wir folgende Matrizen mit r Zeilen und Spalten.

$$S_r = \begin{pmatrix} s_1 & s_2 & \ldots & s_r \\ s_2 & s_3 & \ldots & s_{r+1} \\ \vdots & \vdots & & \vdots \\ s_r & s_{r+1} & \ldots & s_{2r-1} \end{pmatrix}$$

Die Einträge sind als Syndromwerte des empfangenen Worts v alle bekannt. Schaut man sich nun die ersten t Gleichungen von (7.2) genau an, mit den t Unbestimmten q_i und den Koeffizienten s_j, dann stellt man fest, dass das Matrixschema S_t als Einträge genau diese Koeffizienten hat. Nun brauchen wir aber leider noch etwas mehr an Vorkenntnis, wir formulieren das aber zunächst einmal vage: Aus den Gleichungen (7.1) liest man nämlich ab, dass sich die Matrix S_t aus zwei **Vandermonde-Matrizen** und einer **Diagonalmatrix** mit den Diagonaleinträgen $f_m \neq 0, m \in Tr(f)$ *zusammensetzt*. Damit weiß man auch hier, dass die Spaltenvektoren von S_t linear unabhängig sind und das Gleichungssystem (7.2) mit t Gleichungen und t Unbestimmten eindeutig lösbar ist.

Hier ist das formale Argument. S_t lässt sich nämlich als Matrixprodukt der folgenden drei Matrizen schreiben.

$$\begin{pmatrix} \beta^{m_1} & \dots & \beta^{m_t} \\ \beta^{2m_1} & & \beta^{2m_t} \\ \vdots & & \vdots \\ \beta^{tm_1} & \dots & \beta^{tm_t} \end{pmatrix} \begin{pmatrix} f_{m_1} & & & \\ & f_{m_2} & & \\ & & \ddots & \\ & & & f_{m_t} \end{pmatrix} \begin{pmatrix} 1 & \beta^{m_1} & \dots & \beta^{(t-1)m_1} \\ \vdots & \vdots & & \vdots \\ 1 & \beta^{m_t} & \dots & \beta^{(t-1)m_t} \end{pmatrix}$$

Damit ist dessen Determinante das Produkt der drei einzelnen Determinanten und folglich ebenfalls $\neq 0$. Also hat auch die Produktmatrix maximalen Rang t.

Die eindeutige Lösung liefert also – wie gewünscht – unser Fehlerortungspolynom $q(x) = 1 + q_1 x + \ldots + q_t x^t$.

Wenn es uns nun noch gelingt, auch t zu bestimmen, dann haben wir den letzten Puzzlestein gelegt. Ist aber $t < r \leq e$, dann zeigt unser Gleichungssystem (7.2), dass der r. Spaltenvektor von S_r linear abhängig von den restlichen Spaltenvektoren ist. Die Fehleranzahl t ist daher eindeutig bestimmt als die größte Zahl r, für die die Matrix S_r linear unabhängige Spaltenvektoren hat.

7.4.2 PGZ-Decodierung – Wie der Algorithmus funktioniert

Wir übersetzen unsere Herleitung des Algorithmus in klare Handlungsanweisungen und benutzen dabei die Bezeichnungen aus dem vorangegangenen Punkt.

- Zu einem empfangenen Wort $v = (v_0, v_1, \ldots, v_{n-1})$ berechne das Syndrom $(s_1, \ldots, s_{d-1})$. Ist das Syndrom 0, so ist $v = c$ ein Codewort und wir sind fertig.
- Stelle die **Syndrommatrix** S_e auf

$$S_e = \begin{pmatrix} s_1 & s_2 & \dots & s_e \\ s_2 & s_3 & \dots & s_{e+1} \\ \vdots & \vdots & & \vdots \\ s_e & s_{e+1} & \dots & s_{2e-1} \end{pmatrix}.$$

- Bestimme durch schrittweise Verkleinerung der Matrix S_e um jeweils eine Zeile und Spalte die Zahl t maximal derart, dass die Matrix S_t linear unabhängige Spalten besitzt. Findet man kein solches t, dann beende die Decodierung für dieses v (und markiere v ggf. entsprechend).
- Berechne die Koeffizienten des **Fehlerortungspolynoms** $q(x) = 1 + q_1 x + \ldots + q_t x^t$ als eindeutige Lösung der t Gleichungen mit t Unbekannten $-s_i = q_1 s_{i-1} + \ldots + q_t s_{i-t}$ für $i = t+1, \ldots, 2t$.

- Bestimme die Nullstellen von $q(x)$ durch sukzessives Einsetzen der $1, \beta, \beta^2, \ldots, \beta^{n-1}$. Genau im Falle von $q(\beta^{n-m}) = 0$ ist m Fehlerposition und man erhält so sukzessive $Tr(f)$.
- Berechne schließlich den Fehler $f = (f_0, \ldots, f_{n-1})$ aus dem Gleichungssystem $s_i = \sum_m \beta^{mi} f_m$, wobei die Summe über $m \in Tr(f)$ läuft und $i = 1, \ldots, t$.
- Korrigiere v zu $v - f$.

Dieses Verfahren funktioniert auch für primitive BCH-Codes im engeren Sinne, obwohl wir uns bei der Herleitung auf Reed-Solomon-Codes beschränkt hatten. Bei BCH-Codes ist allerdings eine Verallgemeinerung des ersten Schritts notwendig. Wir haben nämlich bei Reed-Solomon-Codes mit dem *üblichen* Syndrom des Wortes v argumentiert, berechnet mit der Kontrollmatrix H. Im Allgemeinen muss man aber anstelle des n-Tupels v dessen Polynomdarstellung $v(x)$ und anstelle der *üblichen* Syndromwerte das Wort $v(x)$ ausgewertet an den Stellen $\beta, \beta^2, \ldots, \beta^{d-1}$ nehmen, wobei β die zum BCH-Code gehörige primitive n. Einheitswurzel ist. Die zu bestimmenden Fehlerpositionen beziehen sich dann auf die x-Potenzen des Polynoms $v(x)$ in aufsteigender Reihenfolge.

Dividiert man $v(x)$ durch das BCH-Generatorpolynom $g(x)$ mit Rest, dann erhält man bekanntlich das Syndrompolynom $s_v(x)$, d. h. $s_v(x) = v(x) (\mathrm{mod}(g(x))$. Man beachte, dass dabei $v(\beta^i) = s_v(\beta^i)$ gilt. Bei Reed-Solomon-Codes stimmen diese Werte *zufälligerweise* mit dem *üblichen* Syndrom überein.

Bei binären BCH-Codes ist natürlich der zweitletzte Schritt überflüssig. In jedem Fall handelt es sich aber um eine BD-Decodierung.

7.4.3 Beispiel: PGZ-Decodierung

Jetzt zu zwei ausführlichen Beispielen, zunächst ein ganz kleines zur Einstimmung.

Reed-Solomon-Code $RS_5(3)$
Der Reed-Solomon-Code $RS_5(3)$ hat Länge $n = 4$, Minimalabstand $d = 3$ und ist daher 1-fehlerkorrigierend, also $e = 1$. Es ist $\beta = 2$ eine primitive 4. Einheitswurzel in $K = \mathbb{Z}_5$ und die Kontrollmatrix lautet:

$$H = \begin{pmatrix} 1 & \beta & \beta^2 & \beta^3 \\ 1 & \beta^2 & \beta^4 & \beta^6 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 4 & 3 \\ 1 & 4 & 1 & 4 \end{pmatrix}.$$

Es werde das Wort $v = (3, 2, 2, 1) \in (\mathbb{Z}_5)^4$ empfangen und es sei nur ein Fehler aufgetreten. Wir gehen nun Schritt für Schritt den PGZ-Algorithmus durch.

Das Syndrom der Skalarprodukte von v mit den Zeilenvektoren von H ergibt $(s_1, s_2) = (3, 2)$.

Für die Syndrommatrix S_e ergibt sich $S_1 = (s_1) = (3)$.

Bei S_1 gibt's nichts zu verkleinern und der eine Spaltenvektor (3) von S_1 ist linear unabhängig, also folgt $t = 1$.

Wir müssen jetzt das Gleichungssystem $-s_2 = s_1 q_1$ lösen, also eine Gleichung mit einer Unbestimmten q_1. Die Syndromwerte eingesetzt ergibt $-2 = 3q_1$, also $q_1 = (-2)3^{-1} = 3 \cdot 2 = 1$. Das Fehlerortungspolynom ist daher $q(x) = 1 + x$.

Wir bestimmen jetzt die Nullstellen von $q(x) = 1+x$, und zwar in Gestalt der Potenzen β^{4-m} der primitiven 4. Einheitswurzel $\beta = 2$. Die eine Nullstelle $-1 = 4$ von $q(x)$ lässt sich offenbar schreiben als $4 = 2^2 = 2^{4-2}$. Also ist $m = 2$ die eine Fehlerposition von v. Diese ist aber an der 3. Stelle von v, da die Komponenten von v und f mit 0 beginnend nummeriert waren.

Wir berechnen nun den Fehlervektor $f = (0, 0, f_2, 0)$ aus dem Gleichungssystem $s_1 = \beta^2 f_2$, also einer Gleichung mit einer Unbestimmten f_2. Konkret heißt das $3 = 4f_2$, also $f_2 = 3 \cdot 4^{-1} = -3 = 2$. Also ist $f = (0, 0, 2, 0)$ der Fehlervektor.

Wir korrigieren somit v in $v - f = (3, 2, 0, 1)$. Durch Multiplikation mit den Zeilen von H kann man jetzt auch verifizieren, dass $v - f$ wirklich ein Codewort ist.

Obwohl das Beispiel ja sehr einfach ist, stellen wir schon fest, wie umfangreich diese Rechnungen bei größeren Beispielen sein müssen.

Binärer BCH-Code mit Parameter [15, 5]

Jetzt ein praxisnäheres Beispiel über $\mathbb{Z}_2$. Wir betrachten nämlich den binären BCH-Code C von Länge $n = 15$, Dimension $k = 5$ und mit Generatorpolynom $g(x) = x^{10} + x^8 + x^5 + x^4 + x^2 + x + 1$, der für die Codierung der Formatbits des QR-Codes verwendet wird. Der Minimalabstand von C ist 7, er ist also 3-fehlerkorrigierend, d. h. $e = 3$. Außerdem werden wir auch unsere primitive 15. Einheitswurzel β benötigen, für die ja $\beta^4 = \beta + 1$ gilt. Für unser Beispiel betrachten wir die Formatbits $(1, 0, 1, 1, 0)$, die auf der QR-Matrix als Codewort $c = (1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0)$ oder in Polynomschreibweise $c(x) = x^{14} + x^{12} + x^{11} + x^8 + x^4 + x^3 + x^2 + x$ abgebildet werden. Wir nehmen aber an, dass der Barcodescanner stattdessen die Bitfolge $v = (1, 0, 1, \underline{\mathbf{0}}, 0, 0, 1, 0, 0, 0, 1, \underline{\mathbf{0}}, 1, 1, 0)$ gelesen hat. Es sind also 2 Fehler aufgetreten, die fett unterstrichen hervorgehoben sind. In Polynomschreibweise bedeutet das, dass $v(x) = x^{14} + x^{12} + x^8 + x^4 + x^2 + x$ gelesen wurde.Wir wollen mit dem PGZ-Algorithmus decodieren und gehen wieder Schritt für Schritt vor.

Wie oben schon erwähnt, ist hier nicht das *übliche* Syndrom eines linearen Codes zu berechnen, sondern $v(x)$ ausgewertet an $\beta, \beta^2, \ldots, \beta^6$. Benutzt man nun wiederholt $\beta^4 = \beta + 1$, so ergibt sich nach einiger Rechnerei $s_1 = v(\beta) = \beta^5$, $s_2 = v(\beta^2) = \beta^{10}$, $s_3 = v(\beta^3) = \beta$, $s_4 = v(\beta^4) = \beta^5$, $s_5 = v(\beta^5) = \beta^5$ und $s_6 = v(\beta^6) = \beta^2$.

Die Syndrommatrix lautet also

$$S_3 = \begin{pmatrix} \beta^5 & \beta^{10} & \beta \\ \beta^{10} & \beta & \beta^5 \\ \beta & \beta^5 & \beta^5 \end{pmatrix}.$$

Man rechnet nun nach, dass $(\beta^5, \beta^{10}, \beta) = \beta^6(\beta^{10}, \beta, \beta^5) + \beta(\beta, \beta^5, \beta^5)$, also sind die drei Spalten von S_3 linear abhängig. Wir verkleinern daher die Matrix auf

$$S_2 = \begin{pmatrix} \beta^5 & \beta^{10} \\ \beta^{10} & \beta \end{pmatrix}.$$

Diese beiden Spalten sind nun aber linear unabhängig, somit ist die Fehleranzahl $t = 2$.

Wir müssen nun die beiden Gleichungen $-s_3 = q_1 s_2 + q_2 s_1$ und $-s_4 = q_1 s_3 + q_2 s_2$ mit den Unbestimmten q_1 und q_2 lösen. Einsetzen der s_i ergibt konkret $\beta = q_1\beta^{10} + q_2\beta^5$ und $\beta^5 = q_1\beta + q_2\beta^{10}$, folglich $q_1 = \beta^5$ und $q_2 = \beta^{14}$. Das Fehlerortungspolynom ist daher $q(x) = 1 + \beta^5 x + \beta^{14}x^2$.

Nun müssen wir die Nullstellen von $q(x)$ in Form β^{15-m} durch sukzessives Ausprobieren bestimmen. Es sind dies $\beta^4 = \beta^{15-11}$ und $\beta^{12} = \beta^{15-3}$. Die Fehler befinden sich daher an der 3. und 11. x-Potenz des gelesenen Wortes $v(x)$, d. h. an der 4. Stelle von vorn und von hinten.

Da es sich um einen binären Code handelt, korrigieren wir also $v(x)$ bzw. v, indem wir an den beiden oben genannten Stellen das Bit ändern, nämlich beide Male von 0 nach 1.

Binäre BCH-Codes sind also eigentlich nur *verkappte binäre* Codes. Beim Decodieren mit dem PGZ-Algorithmus muss man dennoch im *großen* Körper rechnen, in unserem Fall also im Körper mit $2^4 = 16$ Elementen.

7.5 Berlekamp-Massey-Algorithmus

7.5.1 Komplexität: PGZ-Algorithmus – aus historischer Sicht

Wir starten mit einer kleinen Analyse der Komplexitäten der einzelnen Schritte des PGZ-Algorithmus und stellen gleichzeitig die wesentlichsten Optimierungspotenziale vor.

Die ersten beiden Schritte zur Berechnung der Syndromwerte und der Syndrommatrix sind algorithmisch einfach, da es sich hierbei entweder nur um einige Skalarprodukte oder um Auswertungen von Polynomen handelt.

Die Schritte Nummer drei und vier sind komplex. Zwar würde man – mit Methoden der linearen Algebra – die lineare Unabhängigkeit der Spaltenvektoren besser mittels sog. **Determinanten** entscheiden. Aber auch dies ist für S_e in der Größenordnung von $\sim e!$ elementaren Operationen. Jedenfalls werden dieser dritte Schritt und die anschließende Lösung des linearen Gleichungssystems im vierten Schritt, z. B. mit dem **Gauß'schen Eliminationsverfahren** (Komplexität $\sim e^3 + e^2$), bei steigender Fehlerkorrekturkapazität extrem rechenaufwendig, sodass *gute* Codes mit *reinen* PGZ-Decodierern in der erforderlichen Zeit kaum decodiert werden können. Der amerikanische Mathematiker **Elwyn Berlekamp** (geb. 1940), den wir in Abschn. 3.4 schon im Zusammenhang mit der Kommentierung der Golay-Publikation erwähnt hatten, hat 1965 einen iterativen Decodierungsalgorithmus vorgestellt, der dieses Problem mit sehr viel geringerem Re-

chenaufwand löst und dadurch den Einsatz von BCH-Codes in Kommunikationssystemen erlaubte. **James Lee Massey** (1934–2013), lange Jahre Professor für Kryptografie an der ETH Zürich, erkannte 1969, dass man diesen Algorithmus als *adaptives* Schieberegister interpretieren kann, weshalb er heute als **Berlekamp-Massey-Algorithmus** bezeichnet wird. Wir werden diesen Algorithmus im Anschluss vorstellen.

Der fünfte Schritt – nämlich das Bestimmen der Nullstellen des Fehlerortungspolynoms – ist grundsätzlich kein Problem, zumal **Robert T. Chien** 1964 eine Schieberegisterschaltung vorgelegt hat, mit der man diesen Test besonders effizient durchführen kann. Dieses recht einfache Verfahren ist als **Chien Search** bekannt. Die Details hierzu findet man am Ende dieses Abschnitts.

Der sechste Schritt letztlich, der für nichtbinäre Codes und damit insbesondere für Reed-Solomon-Codes zur Berechnung des Fehlervektors notwendig ist, erfordert ebenfalls die Lösung eines vergleichbar großen linearen Gleichungssystems. Seit 1969 ist aber auch dies mit dem sog. **Forney-Algorithmus** effizient lösbar. Der amerikanische Elektroingenieur **George David Forney** (geb. 1940) hat viele weitere bedeutende Beiträge im Bereich der Codierungs- und Informationstheorie geliefert und wird uns noch wiederholt in Kap. 9 im Zusammenhang mit Faltungscodes begegnen. Auf den Forney-Algorithmus werden wir hier allerdings nicht eingehen, zumal wir in Abschn. 7.6 noch einen ganz anderen Ansatz, nämlich die Decodierung mittels **euklidischen Algorithmus**, vorstellen werden. Insgesamt stellt also heute das Decodieren auch von größeren Reed-Solomon- und BCH-Codes kein Problem mehr dar. Allein mit dem Berlekamp-Massey-Algorithmus reduziert sich die Komplexität von Schritt drei und vier auf $\sim 6e^2$ elementare Operationen.

7.5.2 Grundzüge des Berlekamp-Massey-Algorithmus

Wir wollen hier nur die wesentlichen Ideen und Schritte des berühmten **Berlekamp-Massey-Algorithmus** herausarbeiten. Zur Erinnerung nochmals der Sachverhalt.

Die Syndromwerte $s_1, \ldots, s_{d-1}$ sind bekannt, es ist $t \leq e$ und $2e + 1 \leq d$ und es gilt das Gleichungssystem $s_i = -q_1 s_{i-1} - \ldots - q_t s_{i-t}$ für $i = t + 1, \ldots, d - 1$.

Es geht in Schritt drei und vier beim PGZ-Algorithmus darum, dasjenige $t \leq e$ zu bestimmen, für das das Gleichungssystem aus den ersten t Gleichungen genau eine Lösung $(q_1, \ldots, q_t)$ hat, und diese Lösung dann auch zu berechnen. Wir haben erwähnt, dass es dazu durchaus allgemeine Methoden gibt, aber deren Komplexität für die Anwendung zu groß ist.

Der **Berlekamp-Massey-Algorithmus** beruht im Wesentlichen auf zwei Ideen:

- Wir hatten uns im letzten Abschnitt mit den Syndrommatrizen S_r sozusagen *von oben* an t angenähert. Ausgehend von S_e haben wir die Matrizen S_r so lange schrittweise verkleinert, bis wir bei der Matrix S_t angelangt waren, die t linear unabhängige Spalten besitzt und bei der das zugehörige Gleichungssystem dann eine eindeutige Lösung hat.

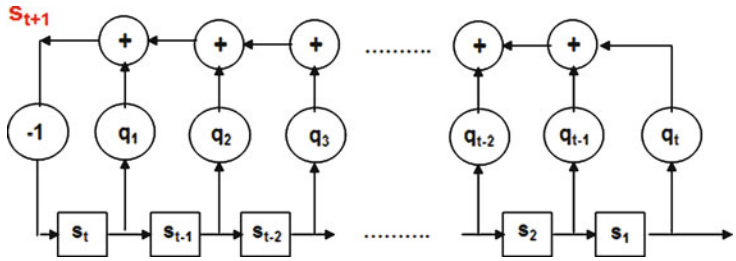


Abb. 7.5 Rückgekoppeltes Schieberegister im Berlekamp-Massey-Algorithmus

Im Berlekamp-Massey-Algorithmus nähert man sich sozusagen t *von unten* an, wie wir gleich sehen werden.

- Unser Gleichungssystem ist allerdings auch kein ganz allgemeines System, auf das man dann auch nur allgemeine Lösungsmethoden anwenden könnte, sondern ein ganz spezielles, nämlich ein **rekursives**: Der Wert s_i hängt über eine **Rekursionsgleichung** jeweils von den t Vorgängerwerten $s_{i-1}, \ldots, s_{i-t}$ ab. So etwas *schreit* geradezu nach Schieberregistern. Es handelt sich dabei genau genommen um ein sog. **linear rückgekoppeltes Schieberegister**, das aber im Gegensatz zu den Schieberregistern, die wir in Abschn. 6.2 zur Multiplikation und Division mit einem Polynom $g(x)$ verwendet hatten, ohne Inputdaten auskommt. Es werden dabei alle Zellen zur Berechnung des nächsten Wertes *mit einer linearen Gleichung rückgekoppelt*. In Abb. 7.5 ist der erste Schritt für $i = t + 1$ zu sehen.

Man beschreibt ein linear rückgekoppeltes Schieberegister gern kurz mit dem sog. **Rückkopplungspolynom** in einer Unbestimmten z, nämlich $C(z) = 1 + q_1 z + q_2 z^2 + \ldots + q_t z^t$, welches also genau die Skalierer q_j des Schieberegisters als Koeffizienten hat.

Das Schieberegister beginnt mit der initialen Vorbelegung der Zellen $s_1, \ldots, s_t$ und rechnet mit jedem Takt das nächste Glied der Rekursion aus bis zu s_{d-1}, während dabei jeweils ein Wert nach rechts als Output herausgeschoben wird. Sind wir damit jetzt fertig? Natürlich nicht, denn wir kennen zwar die Struktur des Schieberegisters, aber weder die q_j noch seine Länge t. Die Fragestellung ist also anders herum: Man konstruiere das kürzeste derartige Schieberegister, dessen Länge dann das gesuchte t ist, und bei dem nach Initialisierung durch die $s_1, \ldots, s_t$ die restlichen $s_{t+1}, \ldots, s_{d-1}$ berechnet werden.

Operativ geht man also iterativ vor und nähert sich dabei mit Schiebregistern der Länge r sukzessive dem gesuchten t von unten her an. Man startet dabei mit einem Schieberegister der Länge $r = 0$, das eine Outputsequenz von lauter Nullen erzeugt. Wir führen mal konkret die ersten Schritte der Iteration aus. Die Schieberegisterlänge r steht also

auf 0. Damit liest sich die erste Gleichung als $s_1 = 0$. Diese muss man jetzt auf Gültigkeit überprüfen. Ist also s_1 wirklich gleich 0, so ist die Gleichung erfüllt und man geht zur Prüfung der zweiten Gleichung $s_2 = 0$ über, also weiterhin mit Länge $r = 0$. Ist dagegen $s_1 \neq 0$, so verlängert man das Schieberegister auf $r = 1$ und die erste Gleichung liest sich als $s_2 = -q_1 s_1$. Wählen wir hier $q_1 = -s_2 s_1^{-1}$, so ist diese erfüllt und wir gehen weiterhin mit $r = 1$ zur zweiten Gleichung $s_3 = -q_1 s_2 = s_2 s_1^{-1} s_2$ über, die es jetzt zu prüfen gilt. Wie sieht's nun allgemein für den Schritt i aus?

Wir nehmen also an, dass wir bis hierher ein Schieberegister der Länge r mit Rückkopplungspolynom $C(z) = 1 + q_1 z + \ldots + q_r z^r$ konstruiert haben, wobei $C(z)$ genau die Skalierer des bisher konstruierten Schieberegisters als Koeffizienten hat. Es gilt nun zu untersuchen, ob die sog. **Diskrepanz** $\Delta = s_i + q_1 s_{i-1} + \ldots + q_r s_{i-r}$ gleich oder ungleich 0 ist.

- Ist $\Delta = 0$, so macht man mit demselben r und demselben $C(z)$ für $i + 1$ weiter.
- Ist dagegen $\Delta \neq 0$, so muss man Korrekturen vornehmen. Sei dazu j *der* Schritt in der Iteration, bei dem zum letzten Mal eine Verlängerung des Schieberegisters vorgenommen wurde. Sei weiterhin Δ_j die damalige Diskrepanz und $C_j(z)$ das Rückkopplungspolynom vor dem Schritt j.
  - Sofern $2r \leq i - 1$ ist, verlängert man r auf $i - r$.
  - Als neues Rückkopplungspolynom wählt man $C(z) - \Delta \Delta_j^{-1} z^{i-j} C_j(z)$.

Das Verfahren endet mit der letzten Rekursionsgleichung für s_{d-1}, womit dann auch die gesuchte Länge $r = t$ und das gesuchte Rückkopplungspolynom $C(z) = 1 + q_1 z + \ldots + q_t z^t$ bestimmt sind.

Dass dies so ist, muss man natürlich auch formal nachweisen. Wir wollen darauf allerdings nicht näher eingehen (s. [Wil, Kap. 9]).

7.5.3 Anwendung: Stromchiffre

Der Berlekamp-Massey-Algorithmus war nicht nur ein wichtiger Mosaikstein bei der PGZ-Decodierung. Mit ihm kann nämlich ganz allgemein bestimmt werden, was das kürzeste, linear rückgekoppelte Schieberegister ist, welches eine gegebene Wertefolge erzeugt. Die Länge dieses Schieberegisters wird als **lineare Komplexität** der Sequenz bezeichnet.

In der Kryptografie verwendet man bei symmetrischen Chiffrierverfahren binäre Pseudozufallsfolgen, die man zum Verschlüsseln auf die Informationsbitfolge addiert. Man nennt ein solches Verfahren auch **Stromchiffre**. Es geht auf **Gilbert Vernam** ins Jahr 1918 zurück. Diese Pseudozufallsfolgen generiert man in der Regel mittels linear rück-

gekoppelter Schieberegister. Wenn ein potenzieller Angreifer aber einmal das Bildungsgesetz der Pseudozufallsfolge erkannt hat, so kann er den gesamten Nachrichtenverkehr ungehindert mithören. Es ist daher zur Bewertung der Sicherheit von Stromchiffren von höchstem Interesse, die lineare Komplexität der Sequenz zu kennen. Je höher diese ist, desto schwieriger und langwieriger ist der Versuch, das Chiffrierverfahren zu *knacken*.

7.5.4 Beispiel: Berlekamp-Massey-Algorithmus

Linerare Komplexität einer Bitfolge
Hier also zunächst ein Beispiel für den Berlekamp-Massey-Algorithmus mit der binären Folge $(s_1, \ldots, s_7) = (0, 0, 1, 1, 0, 1, 1)$.

Startwerte: $r = 0$, $j = 0$, $C_0(z) = 1$, $\Delta_0 = 1$.

- $i = 1$: Es ist $\Delta = s_1 = 0$ und weiter.
- $i = 2$: Es ist $\Delta = s_2 = 0$ und weiter.
- $i = 3$: Es ist $\Delta = s_3 = 1$. Wegen $2r = 0 \leq 2 = i - 1$ setzen wir $r = i - 0 = 3$ und $C(z) = 1 + z^3 C_0(z) = 1 + z^3$. Außerdem $j = 3$, $C_3(z) = 1$ und $\Delta_3 = 1$.
- $i = 4$: Es ist $\Delta = s_4 + q_1 s_3 + q_2 s_2 + q_3 s_1 = 1 + 0 + 0 + 0 = 1 \neq 0$. Wegen $2r = 6 > 3 = i - 1$ bleibt $r = 3$, und $C(z) = 1 + z^3 + zC_3(z) = 1 + z + z^3$. Es bleiben auch $j = 3$, $C_3(z) = 1$ und $\Delta_3 = 1$.
- $i = 5$: Es ist $\Delta = s_5 + q_1 s_4 + q_2 s_3 + q_3 s_2 = 0 + 1 + 0 + 0 = 1 \neq 0$. Wegen $2r = 6 > 4 = i - 1$ bleibt $r = 3$ und $C(z) = 1 + z + z^3 + z^2 C_3(z) = 1 + z + z^2 + z^3$. Es bleiben $j = 3$, $C_3(z) = 1$ und $\Delta_3 = 1$.
- $i = 6$: Es ist $\Delta = s_6 + q_1 s_5 + q_2 s_4 + q_3 s_3 = 1 + 0 + 1 + 1 = 1 \neq 0$. Wegen $2r = 6 > 5 = i - 1$ bleibt $r = 3$ und $C(z) = 1 + z + z^2 + z^3 + z^3 C_3(z) = 1 + z + z^2$. Es bleiben $j = 3$, $C_3(z) = 1$ und $\Delta_3 = 1$.
- $i = 7$: Es ist $\Delta = s_7 + q_1 s_6 + q_2 s_5 + q_3 s_4 = 1 + 1 + 0 + 0 = 0$ und fertig.

Das gesuchte Schieberegister hat also Länge $t = 3$ sowie die Skalierer $q_1 = q_2 = 1$ und $q_3 = 0$. Die lineare Komplexität der Folge ist somit 3 und Abb. 7.6 visualisiert den gesamten Schaltplan.

Binärer BCH-Code mit Parameter [15, 5]
Ein anspruchsvolleres Beispiel ist die Sequenz der Syndromwerte unseres binären BCH-Codes C, für den wir im letzten Abschnitt die PGZ-Decodierung durchgeführt haben. Die

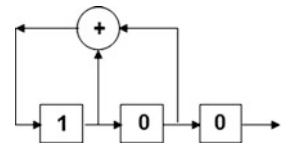


Abb. 7.6 Kürzestes rückgekoppeltes Schieberegister zu $(0, 0, 1, 1, 0, 1, 1)$

Sequenz lautete hierfür $(s_1, \ldots, s_6) = (\beta^5, \beta^{10}, \beta, \beta^5, \beta^5, \beta^2)$ und wir starten wieder den Berlekamp-Massey-Algorithmus.

Startwerte: $r = 0$, $j = 0$, $C_0(z) = 1$, $\Delta_0 = 1$.

- $i = 1$: Es ist $\Delta = s_1 = \beta^5 \neq 0$. Wegen $2r = 0 \leq 0 = i - 1$ setzen wir $r = i - 0 = 1$ und $C(z) = 1 + z^{i-j}\Delta\Delta_0^{-1}C_0(z) = 1 + \beta^5 z$. Außerdem $j = 1$, $C_1(z) = 1$ und $\Delta_1 = \beta^5$.
- $i = 2$: Es ist $\Delta = s_2 + q_1 s_1 = \beta^{10} + \beta^5\beta^5 = 0$ und weiter.
- $i = 3$: Es ist $\Delta = s_3 + q_1 s_2 = \beta + \beta^5\beta^{10} = \beta + 1 = \beta^4 \neq 0$. Wegen $2r = 2 \leq 2 = i - 1$ setzen wir $r = i - 1 = 2$ und $C(z) = 1 + \beta^5 z + z^{i-j}\Delta\Delta_1^{-1}C_1(z) = 1 + \beta^5 z + \beta^4\beta^{-5}z^2 = 1 + \beta^5 z + \beta^{14}z^2$. Außerdem $j = 3$, $C_3(z) = 1 + \beta^5 z$ und $\Delta_3 = \beta^4$.

Wir überlassen die restlichen Schritte dem Leser. Es sollte sich folgendes Ergebnis einstellen.

- $i = 4$: $r = 2$ und $C(z) = 1 + \beta^5 z + \beta^{14}z^2$
- $i = 5$: $r = 2$ und $C(z) = 1 + \beta^5 z + \beta^{14}z^2$
- $i = 6$: $r = 2$ und $C(z) = 1 + \beta^5 z + \beta^{14}z^2$

Also ist $t = 2$ und das Fehlerortungspolynom ist $q(x) = 1 + \beta^5 x + \beta^{14}x^2$, so wie wir es bei der PGZ-Decodierung noch ohne Berlekamp-Massey-Algorithmus bereits bestimmt hatten.

7.5.5 Chien Search

Zum Schluss dieses Abschnitts – sozusagen als Ausklang – die **Chien Search**. Es gilt dabei, das Fehlerortungspolynom $q(x) = 1 + q_1 x + \ldots + q_t x^t$ an den Potenzen β^i der primitiven n. Einheitswurzel β auszuwerten, also $q(\beta^i) = 1 + q_1\beta^i + \ldots + q_t(\beta^i)^t = b_{0,i} + b_{1,i} + \ldots + b_{t,i}$ zu bestimmen. Die Idee ist, dass auch $q(\beta^{i+1}) = 1 + q_1\beta^{i+1} + \ldots + q_t(\beta^{i+1})^t = 1 + q_1\beta^i\beta + \ldots + q_t(\beta^i)^t\beta^t = b_{0,i+1} + b_{1,i+1} + \ldots + b_{t,i+1}$ gilt und sich daraus die rekursive Beziehung $b_{j,i+1} = b_{j,i}\beta^j$ für $i = 0, \ldots, n-1$ und $j = 0, \ldots, t$ ergibt.

Bei der Chien Search wird für jedes i sukzessive überprüft, ob $\sum_j b_{j,i} = 0$ ist, also ob $q(\beta^i) = 0$ gilt. Dabei werden die nächsten $b_{j,i+1}$ jeweils durch obige Rekursion berechnet. Dieses sukzessive Multiplizieren der bereits errechneten $b_{j,i}$ mit β^j erfolgt mittels Schieberegisterschaltung und reduziert die Komplexität der Nullstellenberechnung erheblich gegenüber der naheliegenden Methode, alle Werte β^i der Reihe nach in $q(x)$ einzusetzen und den Funktionswert zu berechnen.

7.6 Justesen-Codes, Goppa-Codes und Euklid-Decodierung

7.6.1 Justesen-Codes

Wir wollen zunächst als Anwendung von Reed-Solomon-Codes noch ein weiteres Codierverfahren vorstellen, die **Justesen-Codes**. Die Idee von **Jørn Justesen**, die er 1972 publizierte, war die folgende. Ein Reed-Solomon-Code $RS_q(d)$ der Länge $n = q - 1$ und Dimension $k = q - d$ muss ja über großen Körpern K gebildet werden und wir gehen daher von einem Körper mit $q = 2^m$ Elementen aus. Andererseits müssen zur digitalen Datenübertragung die einzelnen Zeichen $c_i \in K$ eines Codeworts $c = (c_0, \ldots, c_{n-1})$ wieder binär dargestellt werden, also als m-Tupel über $\mathbb{Z}_2$. Dann könnte man doch, zusätzlich zum *äußeren* Reed-Solomon-Code, jedes m-Tupel mit einem weiteren *inneren* Code C versehen. Mehr noch, bei Justesen-Codes werden jedes der m-Tupel mit verschiedenen inneren Codes C_i $(i = 0, \ldots, n-1)$ der gleichen Länge codiert.

Wir wollen hier konkret das Standardbeispiel eines **Justesen-Codes** betrachten. Dabei werden die Zeichen $c_i \in K$ mit anderen Körperelementen multipliziert. Welche Körperelemente bieten sich da an? Natürlich die primitive n. Einheitswurzel β und ihre Potenzen β^i. Die Codes C_i werden dabei also definiert durch Multiplikation mit β^i. Zu einem Codewort $c = (c_0, \ldots, c_{n-1}) \in RS_q(d)$ betrachtet man das Matrixschema

$$J = \begin{pmatrix} c_0 & c_1 & c_2 & \ldots & c_{n-1} \\ c_0\beta^0 & c_1\beta & c_2\beta^2 & \ldots & c_{n-1}\beta^{n-1} \end{pmatrix}$$

und interpretiert die Spalten $(c_i, c_i\beta^i)$ als $2m$-Tupel über $\mathbb{Z}_2$. Liest man diese Matrix nun spaltenweise aus, so ergibt sich ein binärer Code der Länge $2mn$ und Dimension mk.

Die schlechte Nachricht ist: Aus einem Reed-Solomon-Code der Rate k/n wurde ein Justesen-Code der Rate $mk/2mn = k/2n$, d. h., die Rate hat sich halbiert. Für den Minimalabstand kann man jedoch eine untere Schranke angeben. Diese lautet in ihrer einfachsten Form: Ist der Minimalabstand d des Reed-Solomon-Codes $d \geq m(2m+1)$, so ist der Minimalabstand d^* des zugehörigen Justesen-Codes $d^* \geq (2m)^2$.

Wir wollen uns das kurz überlegen und dabei zunächst sehen, wie sich der *Twist* in der unteren Matrixhälfte auswirkt. Sind nämlich zwei der c_i in der oberen Matrixhälfte gleich und $\neq 0$, so sind die entsprechenden m-Tupel in der unteren Matrixhälfte verschieden. Daher sind alle Spalten $\neq 0$ in der Matrix J paarweise verschieden. Sei nun $0 \neq c \in RS_q(d)$, dann sind mindestens d der $c_i \neq 0$, also enthält J mindestens d paarweise verschiedene Spalten $\neq 0$. Nach Voraussetzung gilt $d \geq m(2m+1) = \binom{2m}{1} + \binom{2m}{2}$, somit enthält J mindestens $\binom{2m}{1}$ Spalten vom Gewicht mindestens 1 und mindestens $\binom{2m}{2}$ Spalten vom Gewicht mindestens 2. Daher gilt $d^* \geq \binom{2m}{1}1 + \binom{2m}{2}2 = (2m)^2$.

Diese Abschätzung führt allerdings nur dann auf gute Justesen-Codes, wenn man m und damit auch n groß wählt. Für $m = 16$ ist beispielsweise $q = 2^{16} = 65.536$ und $n = 2^{16} - 1 = 65.535$. Wenn wir im Reed-Solomon-Code $RS_{65.536}(d)$ den Minimalabstand $d \geq m(2m+1) = 528$ wählen, so hat der entsprechende Justesen-Code also

Minimalabstand $d^* \geq (2m)^2 = 1024$. Man kann mit der Justesen-Methode also gute Codes erzeugen, aber leider nur für große Blocklängen. Genauer gesagt konnte Justesen eine Folge von $[n_i, k_i, d_i]$-Codes mit wachsender Länge n_i konstruieren, bei denen weder die Rate k_i/n_i noch der relative Minimalabstand d_i/n_i gegen 0 gehen, man sagt dazu auch *asymptotisch gut*. Für Justesen-Codes gibt es leider keine guten Decodieralgorithmen. Sie sind daher im Rückblick gesehen eher von theoretischem Interesse und werden in der Praxis kaum angewandt.

7.6.2 Goppa-Codes

Wir wollen zum Abschluss dieses Kapitels noch zu einer Verallgemeinerung von primitiven BCH-Codes im engeren Sinne kommen und damit auch zu einer zweiten Verallgemeinerung von Reed-Solomon-Codes.

Es sei hierzu wieder $|K| = q$ und $|L| = q^s$. Sei weiterhin $n = q^s - 1$, ε eine primitive n. Einheitswurzel in L und C der BCH-Code über K mit Auslegungsdistanz d. Für $c(x) = c_0 + c_1 x + \ldots + c_{n-1}x^{n-1} \in C$ gilt daher $c_0 + c_1(\varepsilon^j) + \ldots + c_{n-1}(\varepsilon^j)^{n-1} = 0$ für $j = 1, \ldots, d-1$, also $(x^n - 1)\sum_i c_i/(x - \varepsilon^{-i}) = \sum_i c_i \sum_j x^j (\varepsilon^{-i})^{n-1-j} = \sum_j x^j \sum_i c_i (\varepsilon^{j+1})^i = x^{d-1} f(x)$, und folglich $\sum_i c_i/(x - \varepsilon^{-i}) = x^{d-1} f(x)/(x^n - 1)$ mit einem Polynom $f(x) \in L[x]$, wobei alle Summen von 0 bis $n-1$ laufen. Also ist $c(x) = c_0 + c_1 x + \ldots + c_{n-1}x^{n-1} \in K[x]$ genau dann ein Codepolynom im BCH-Code C, wenn man $\sum_i c_i/(x - \varepsilon^{-i})$ schreiben kann als $x^{d-1} f(x)/(x^n - 1)$ mit einem $f(x) \in L[x]$.

Diese Überlegung führte den russischen Mathematiker **Valery Denisovich Goppa** (geb. 1939) im Jahr 1970 zu folgender Verallgemeinerung.

Seien $K \subseteq L$, $|K| = q$, $|L| = q^s$, $g(x) \in L[x]$ vom Grad t und $\Delta = \{\gamma_0, \gamma_1, \ldots, \gamma_{n-1}\}$ paarweise verschiedene Elemente von L für eine natürliche Zahl n. Es sei außerdem $g(\gamma_i) \neq 0$ für alle $i = 0, \ldots, n-1$. Dann umfasst der **Goppa-Code $\Gamma(\Delta, g)$** alle $c = (c_0, c_1, \ldots, c_{n-1}) \in K^n$ mit $\sum_i c_i/(x - \gamma_i) = 0 (\mathrm{mod}\, g(x))$ ist. Dies ist so zu verstehen, dass bei der linken Seite – geschrieben als $a(x)/b(x)$ mit teilerfremden $a(x)$, $b(x)$ – der Zähler ein L[x]-Vielfaches von $g(x)$ ist.

Man beachte hierbei $1/(x - \gamma_i) = (-1/g(\gamma_i))(-g(\gamma_i)/(x - \gamma_i)) = (-1/g(\gamma_i))(g(x) - g(\gamma_i))/(x - \gamma_i) (\mathrm{mod}\, g(x))$. Weil der Nenner $x - \gamma_i$ den Zähler $g(x) - g(\gamma_i)$ teilt, ist folglich $1/(x - \gamma_i)$ ein Polynom in $L[x]$ vom Grad $<$ grad $(g(x))$, modulo $g(x)$ gelesen.

Gleiches gilt auch für $\sum_i u_i/(x - \gamma_i)$ mit $u = (u_0, \ldots, u_{n-1}) \in K^n$ und damit insbesondere für $\sum_i c_i/(x - \gamma_i)$.

7.6.3 Beispiel: Justesen- und Goppa-Codes

Justesen-Code zum Reed-Solomon-Code $RS_4(2)$
Zunächst zu einem kleinen Beispiel für einen Justesen-Code. Wir verwenden den Reed-Solomon-Code $RS_4(2)$, also $|K| = 2^2$, $n = 3$, $d = 2$ und $k = 2$. Es ist $K^* = \{\beta^0 = 1, \beta, \beta^2\}$ mit einer primitiven 3. Einheitswurzel β, $\beta^2 = \beta + 1$ und $RS_4(2)$ hat folgende Generatormatrix.

$$G = \begin{pmatrix} 1 & 1 & 1 \\ 1 & \beta & \beta^2 \end{pmatrix}$$

Das Justesen-Schema sieht dann für ein Codewort $(c_0, c_1, c_2) \in RS_4(2)$ so aus:

$$J = \begin{pmatrix} c_0 & c_1 & c_2 \\ c_0 & \beta c_1 & \beta^2 c_2 \end{pmatrix}.$$

Ist z. B. $(c_0, c_1, c_2) = (0, \beta + 1, \beta)$, binär gelesen $(0, 0, 1, 1, 1, 0)$, so ist das Justesen-Codewort $(0, 0, \beta + 1, 1, \beta, 1)$, binär gelesen also $(0, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1)$.

BCH- und Reed-Solomon-Codes als Goppa-Code
Wir haben bereits hergeleitet, dass primitive BCH-Codes im engeren Sinne Goppa-Codes sind. Dies gilt dann insbesondere auch für Reed-Solomon-Codes für den Spezialfall $K = L$. Hierzu muss man in der Definition von Goppa-Codes nämlich $n = q^s - 1$, $\Delta = \{\gamma_i = \varepsilon^{-i} | 0 \leq i \leq n - 1\} = L^*$ sowie $g(x) = x^{d-1}$ wählen, und beachten, dass $b(x)$ das Polynom $(x^n - 1)$ teilt.

Ein kleiner Goppa-Code
Wir wollen auch noch kurz den kleinsten Goppa-Code erwähnen, der nicht auch gleichzeitig BCH-Codes ist. Hierzu sei zunächst $K = \mathbb{Z}_2$ und $g(x) = x^2 + x + 1$. Da $g(x)$ den Körper mit $2^2 = 4$ Elementen erzeugt, hat $g(x)$ keine Nullstelle im Körper L mit $2^3 = 8$ Elementen. Wir können daher $\Delta = L$ wählen, folglich hat $\Gamma(\Delta, g)$ Länge 8. Kontrollmatrizen von Goppa-Codes lassen sich zwar recht leicht berechnen, dennoch wollen wir hier nicht näher darauf eingehen. Jedenfalls hat $\Gamma(\Delta, g)$ die folgende Kontrollmatrix

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

und ist daher ein binärer $[8, 2, 5]$-Code mit Generatormatrix

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

7.6.4 Güte von Goppa-Codes

Für Goppa-Codes $\Gamma(\Delta, g)$ gelten die folgenden Abschätzungen.

- Dimension $\geq n - \text{st}$.
- Minimalabstand $\geq t + 1$ (**verallgem. BCH-Schranke**).
- Minimalabstand $\geq 2t + 1$, falls $\Gamma(\Delta, g)$ binär und $g(x)$ ohne mehrfache Nullstelle ist.

Die verallgemeinerte BCH-Schranke folgt sofort aus der Definition von Goppa-Codes. Ist nämlich $c = (c_0, c_1, \ldots, c_{n-1}) \neq 0$ ein Codewort vom Gewicht w, so hat – wenn man die linke Seite auf den Hauptnenner bringt – der Zähler $a(x)$ höchstens Grad $w - 1$. Da aber $g(x)$ das Polynom $a(x)$ teilt, folgt sofort $t = \text{grad}\,(g(x)) \leq \text{grad}\,(a(x)) \leq w - 1$. Für den Spezialfall der BCH-Codes ist also der Minimalabstand $\geq t + 1 = d$ und dieser einfache Beweis ergibt somit genau die BCH-Schranke.

Goppa-Codes sind in der Tat so gut, dass sie asymptotisch, also für große Längen n, die Gilbert-Schranke für optimale Codes erreichen (s. dazu Abschn. 1.4). Es gibt auch sehr effiziente Decodieralgorithmen, wie insbesondere die Methode, die auf dem euklidischen Algorithmus beruht und die wir unten skizzieren wollen. Allerdings scheinen Goppa-Codes enttäuschenderweise bislang keinen Eingang in eine gängige Praxisanwendung der Codierungstheorie gefunden zu haben.

Genau genommen handelt es sich bei unserer Definition aber nur um die sog. ***klassischen* Goppa-Codes**. Das eigentliche Verdienst von Goppa war es nämlich, dass er die Anwendungsmöglichkeit von Methoden aus der **algebraischen Geometrie** in der Codierungstheorie erkannte und weiterentwickelte, und zwar in der Folge seiner Publikation der klassischen Codes bis in die 1980er-Jahre. Dies hat dann auch zu reger Forschung und vielen sehr guten Codes geführt.

7.6.5 Anwendung: McEliece-Kryptosystem

Es soll aber nicht unerwähnt bleiben, dass Goppa-Codes in der Kryptografie bei asymmetrischen Chiffren Anwendung finden, nämlich beim **McEliece-Kryptosystem**. Als öffentliche Schlüssel wählt man dabei eine *leicht* modifizierte Generatormatrix eines Goppa-Codes.

Wer sich mit etwas linearer Algebra auskennt: Die Generatormatrix wird dabei mit zwei weiteren Matrizen von links und rechts multipliziert.

Empfohlene Parameter sind dabei aktuell für (n, k, t) mindestens $(1700, 1200, 45)$, perspektivisch jedoch bis zu $(2800, 2000, 65)$. Gegenüber gängigeren Verfahren wie dem **RSA-Algorithmus** ist der McEliece-Algorithmus derzeit wenig praktikabel wegen der großen Menge an Schlüsselparametern. Jedoch ist selbst unter Verwendung von sog. *Quantencomputern* kein effizienter Algorithmus bekannt, der das McEliece-Kryptosystem brechen kann, was es zu einem vielversprechenden Kandidaten für die Zukunft macht.

7.6.6 Decodierung mittels euklidischen Algorithmus

Hier ist also noch ein Codierverfahren, das auf dem **euklidischen Algorithmus** beruht und das nicht nur für Goppa-Codes, sondern dann auch für primitive BCH-Codes im engeren Sinne und Reed-Solomon-Codes funktioniert: Die **Euklid-Decodierung** (**Y. Sugiyama**, 1976). Hier sind zunächst wieder die *Hauptakteure* des Algorithmus.

Sei also $\Gamma(\Delta, g)$ ein Goppa-Code und e maximal derart, dass $2e \leq \text{grad}\,(g(x))$ gilt. Weiter sei $c = (c_0, c_1, \ldots, c_{n-1})$ ein Codewort und $v = (v_0, \ldots, v_{n-1}) = c + f$ das empfangene Wort mit Fehlervektor $f = (f_0, \ldots, f_{n-1}) \in K^n$. Dabei seien höchstens e Fehler aufgetreten, d. h. $wt(f) \leq e$. Man betrachtet dann die folgenden Polynome:

- Das **Syndrompolynom**
  $\zeta(x) \in L[x]$, definiert als $\sum_i f_i/(x - \gamma_i) (\text{mod}\, g(x))$ und $\text{grad}\,(\zeta(x)) < \text{grad}\,(g(x))$.
- Das **Fehlerortungspolynom**
  $\sigma(x) = \prod_i (x - \gamma_i)$, wobei i den Träger $Tr(f)$ von f durchläuft.
- Das **Fehlerauswertungspolynom**
  $\omega(x) = \sum_i f_i \prod_j (x - \gamma_j)$, wobei i den Träger $Tr(f)$ durchläuft und $j \in Tr(f)/\{i\}$.

Wir haben bereits ein Syndrompolynom $s(x)$ (s. Abschn. 6.4) und ein Fehlerortungspolynom $q(x)$ (s. Abschn. 7.4) kennengelernt. Die Verwendung eines Fehlerauswertungspolynoms hingegen ist neu.

Warum der Algorithmus funktioniert:

Wegen $\sum_i f_i/(x-\gamma_i) = \sum_i v_i/(x-\gamma_i) (\text{mod}\, g(x))$ ist $\zeta(x)$ direkt aus v berechenbar. Die Polynome $\sigma(x)$ und $\omega(x)$ hingegen berechnet man mittels euklidischen Algorithmus. Sei dazu $r_0(x) = g(x)$ und $r_1(x) = \zeta(x)$. Dann dividiert man zunächst diese beiden Polynome mit Rest, also $r_0(x) = q_2(x)r_1(x) + r_2(x)$ mit geeigneten $q_2(x)$ und $r_2(x) \in L[x]$ und $\text{grad}\,(r_2(x)) < \text{grad}\,(r_1(x))$. Dies iterativ weitergeführt, also $r_i(x) = q_{i+2}(x)r_{i+1}(x) + r_{i+2}(x)$ mit $\text{grad}\,(r_{i+2}(x)) < \text{grad}\,(r_{i+1}(x))$ ist nichts anderes als der euklidische Algorithmus. Dies zeigt zunächst $r_{i+2}(x) = r_i(x) - q_{i+2}(x)r_{i+1}(x)$. Mit

einem Induktionsargument folgert man leicht $r_i(x) = a_i(x)g(x) + b_i(x)\zeta(x)$ für geeignete $a_i(x), b_i(x) \in L[x]$.

Man kann nun Folgendes zeigen (was wir hier aber nicht tun wollen, s. dazu [Mat, Abschn. 10]): Es gibt einen Index k mit $\text{grad}\,(r_k(x)) < e \leq \text{grad}\,(r_{k-1}(x))$, für den man $\sigma(x)$ und $\omega(x)$ an $r_k(x) = a_k(x)g(x) + b_k(x)\zeta(x)$ ablesen kann als $\sigma(x) = b_k(x)/b$ und $\omega(x) = r_k(x)/b$ mit dem höchsten Koeffizienten b von $b_k(x)$.

Wie das Dekodieren mit dem euklidischen Algorithmus funktioniert:

- Zur empfangenen Nachricht $v = (v_0, \ldots, v_{n-1})$ konstruiere man das Syndrompolynom $\zeta(x) \in L[x]$ mit $\zeta(x) = \sum_i v_i/(x - \gamma_i) (\text{mod}\, g(x))$ und $\text{grad}\,(\zeta(x)) < \text{grad}\,(g(x))$.
- Führe den euklidischen Algorithmus mit den Startpolynomen $r_0(x) = g(x)$ und $r_1(x) = \zeta(x)$ aus und berechne sukzessive $r_i(x) = a_i(x)g(x) + b_i(x)\zeta(x)$.
- Sobald das k erreicht wurde mit $\text{grad}\,(r_k(x)) < e \leq \text{grad}\,(r_{k-1}(x))$, lies die Polynome $\sigma(x)$ und $\omega(x)$ ab als $\sigma(x) = b_k(x)/b$ und $\omega(x) = r_k(x)/b$ mit dem höchsten Koeffizienten b von $b_k(x)$.
- Bestimme den Träger $Tr(f)$ von f mittels Nullstellenmenge von $\sigma(x)$.
- Für jedes $i \in Tr(f)$ berechne den Fehlervektor $f = (f_0, \ldots, f_{n-1})$ aus $\omega(\gamma_i) = f_i \prod_j (\gamma_i - \gamma_j)$, wobei j die Menge $Tr(f)/\{i\}$ durchläuft.
- Korrigiere v zu $v - f$.

Für den Spezialfall $g(x) = x^{d-1}$ und $\Delta = \{\gamma_i = \varepsilon^{-i} | 0 \leq i \leq n-1\}$ entspricht das eben beschriebene Verfahren der Decodierung von primitiven BCH-Codes im engeren Sinne und damit auch von Reed-Solomon-Codes. Man beachte, dass der Algorithmus in diesem Fall immer für e maximal mit $2e + 1 \leq d$ funktioniert. Vergleicht man das Verfahren mit dem PGZ-Algorithmus, so stellt man fest, dass die Berechnung

- des Fehlerortungspolynoms (bei PGZ die Optimierung mittels Berlekamp-Massey-Algorithmus) und
- des Fehlerauswertungspolynoms (bei PGZ die Berechnung des Fehlervektors mittels Forney-Algorithmus)

hier allein durch iterative Polynommultiplikation und -division und daher mit schnellen Schieberegisterschaltungen implementierbar sind.

7.6.7 Beispiel: Euklid-Decodierung

Binärer BCH-Code mit Parameter [7, 4]

Wir nehmen zunächst den kleinsten BCH-Code C, den wir konstruiert haben, nämlich binär von Länge $n = 7$ und Auslegungsdistanz $d = 3$ mit Generatorpolynom $f_2(x) = x^3 +$

$x+1$ und einer primitiven 7. Einheitswurzel β mit $\beta^3 = \beta+1$. Dann hat also C das Goppa-Polynom $g(x) = x^2$ und $e = 1$. Das Codewort $c = (c_0, \ldots, c_6) = (1,1,0,1,0,0,0)$ werde gesendet, das Wort $v = (v_0, \ldots, v_6) = (1,1,0,0,0,0,0)$ werde empfangen, wobei also genau ein Fehler aufgetreten ist. Weiter sei $f = v - c = (f_0, \ldots, f_6)$ der Fehlervektor, den es zu bestimmen gilt. Hierzu gehen wir nach obigem Schema vor.

Wir bestimmen das Syndrompolynom $\zeta(x)$ zu v. Es ist $\zeta(x) = \sum_i v_i/(x+\beta^{-i}) = 1/(x+1) + 1(x+\beta^{-1}) = (x^2+1)/(x+1) + (1/\beta^{-2})(x^2+\beta^{-2})/(x+\beta^{-1}) = x + 1 + \beta^2(x+\beta^{-1}) = (1+\beta^2)x + (1+\beta) = \beta^{-1}x + \beta^3 (\mathrm{mod}\, x^2)$.

Nun starten wir den euklidischen Algorithmus mit $g(x) = x^2$ und $\zeta(x) = \beta^{-1}x + \beta^3$ und dividieren dazu zunächst $g(x)$ durch $\zeta(x)$ mit Rest. Dies ergibt $g(x) = (\beta x + \beta^5)\zeta(x) + \beta$.

Hier ist bereits das Ende des Algorithmus erreicht und wir lesen ab $\sigma(x) = (\beta x + \beta^5)/\beta = x + \beta^4$ und $\omega(x) = \beta/\beta = 1$.

$\sigma(x)$ hat als einzige Nullstelle $\beta^4 = \beta^{-3}$, also ist der Fehler an der Stelle $i = 3$ aufgetreten, so wie das Beispiel ja auch konstruiert war. Da der Code binär ist, können wir bereits jetzt zu $v_3 = 1$ korrigieren.

Wegen $1 = \omega(\beta^{-3}) = f_3$ folgt aber auch hieraus formal der Wert 1 für den Fehler an Stelle $i = 3$.

Reed-Solomon-Code $RS_5(3)$

Nun betrachten wir den kleinen Reed-Solomon-Code $RS_5(3)$ mit Parameter $[4,2,3]_5$, der primitiven 4. Einheitswurzel $\beta = 2$ in $K = \mathbb{Z}_5$ und mit Generatorpolynom $f_1(x) = (x-2)(x-4) = x^2+4x+3$. Dann hat also C das Goppa-Polynom $g(x) = x^2$ und $e = 1$. Das Codewort $c = (c_0, \ldots, c_3) = (3,4,1,0)$ werde gesendet, das Wort $v = (v_0, \ldots, v_3) = (3,4,0,0)$ werde empfangen und es soll der Fehlervektor $f = v - c = (f_0, \ldots, f_3)$ bestimmt werden.

Wir bestimmen zunächst wieder das Syndrompolynom $\zeta(x)$ zu v. Nach etwas Rechnung folgt $\zeta(x) = \sum_i v_i/(x-\beta^{-i}) = 3/(x-2^0)+4/(x-2^{-1}) = 3/(x-1)+4/(x-3) = x + 4 (\mathrm{mod}\, x^2)$.

Nun wieder zum euklidischen Algorithmus, wir dividieren also $g(x) = x^2$ durch $\zeta(x) = x+4$ und erhalten $1 = 1g(x) + (4x+4)\zeta(x)$.

Daraus lesen wir ab $\sigma(x) = (4x+4)/4 = x+1$ und $\omega(x) = 1/4 = 4$.

$\sigma(x)$ hat als einzige Nullstelle $-1 = 4 = 2^{-2} = \beta^{-2}$, also ist der Fehler an der Stelle $i = 2$ aufgetreten.

Wegen $4 = \omega(\beta^{-2}) = f_2$ ergibt sich als Fehlervektor $f = (0,0,4,0) = (0,0,-1,0)$, so wie das Beispiel auch konstruiert war.

LDPC-Codes

8

LDPC-Codes, Festplatten und GPS An dieser Stelle greifen wir eine etwas anders gelagerte Idee zur Konstruktion von Codes auf, die **Robert Gray Gallager** bereits im Jahr 1960 vorschlug, die aber in der Folgezeit dem Reed-Solomon- und BCH-*Hype* zum Opfer fiel. Er schlug nämlich vor, speziell Codes mit *dünn* besetzten Kontrollmatrizen zu betrachten (**LDPC**, **Low Density Parity Check**), was in der Regel zu effizienteren Decodieralgorithmen führt. Klare und einheitliche Konstruktionsregeln für solche Codes gab er allerdings nicht an. Wir betrachten einen Ansatz, welcher die Kontrollmatrix mit einem Graphen (sog. **Tanner-Graph**) in Verbindung bringt, sodass man Methoden aus der Graphentheorie anwenden kann. LDPC-Codes erlebten in den späten 1990ern eine Art Revival, wobei man allerdings eher mit **rechnergestützten Pseudozufallsverfahren** die Güte und das Decodierverhalten solcher Codes optimiert. Daher finden LDPC-Codes auch immer mehr Anwendung in der Praxis, z. B. bei **Festplattenlaufwerken** von HGST sowie im neuen Frequenzband L1C der Satellitennavigation **GPS**.

RA-Codes und Bit-Flipping-Decodierung Eine vielbeachtete Familie innerhalb der LDPC-Codes sind die 1998 von **Dariush Divsalar** vorgeschlagenen **RA-Codes (Repeat-Accumulate)** bzw. allgemeiner **IRA-Codes (Irregular Repeat-Accumulate)**. Wir machen uns das Konstruktionsprinzip von RA-Codes klar und können bestätigen, dass sie sicherlich zu den LDPC-Codes gerechnet werden können, obwohl man sie auch gerne als eine Art Zwitter zwischen Blockcodes und Turbo-/Faltungscodes auffasst. Zum Thema Decodierung von LDPC-Codes machen wir uns noch mit dem **iterativen Bit-Flipping-Verfahren** vertraut, einer einfachen Variante eines allgemeineren Vorgehens, bei dem man in jedem Iterationsschritt denjenigen Wert ändert, von dem man am *stärksten* glaubt, dass er falsch ist.

Digitalfernsehen und IRA-Codes Wir bleiben noch beim Thema IRA-Codes, jetzt aber nicht mehr aus theoretischer Sicht, sondern ganz praxisnah. Anfang des Jahrtausends wurde nämlich ein LDPC-Code vom IRA-Typ ausgewählt für den neuen Standard **DVB-**

O. Manz, *Fehlerkorrigierende Codes*, DOI 10.1007/978-3-658-14652-8_8

C2/S2/T2 des **Digitalfernsehens** der 2. Generation. Genauer gesagt wird ein **IRA-Code** als innerer Code genutzt, der außerdem noch von einem äußeren **BCH-Code** als zusätzlicher Schutz umgeben ist. Passend zu jeder Kanalgegebenheit sind die genannten Codes mit unterschiedlichen Raten verfügbar, d. h. zu jeder Rate gibt es einen klar definierten IRA-Code und einen durch sein Generatorpolynom festgelegten BCH-Code. Wir verfolgen die Festlegungen des Standards Schritt für Schritt.

8.1 LDPC-Codes, Festplatten und GPS

8.1.1 Was sind eigentlich LDPC-Codes?

LDPC steht für **Low Density Parity Check** und **LDPC-Code** bedeutet daher auf Deutsch sinngemäß *Code mit dünn besetzten Kontrollmatrizen*, also mit vielen Einträgen gleich 0. Bei LDPC-Codes beschränkt man sich stets auf binäre Codes.

LDPC-Codes werden auch manchmal **Gallager-Codes** genannt, zu Ehren des amerikanischen Elektrotechnikers **Robert Gray Gallager** (geb. 1931), der das Konzept der LDPC-Codes Anfang der 1960er am Massachusetts Institute of Technology (MIT) entwickelte. Allerdings definiert Gallager seine LDPC-Codes nicht allzu konkret, er spricht von *wenigen* Einsen in der Kontrollmatrix, insbesondere aber solche, die bei geeigneter Erweiterung durch Zeilenlinearkombinationen eine feste Anzahl von Einsen in den Zeilen und auch eine feste Anzahl von Einsen in den Spalten haben. Hierzu gibt er das in Abb. 8.1 wiedergegebene Beispiel an.

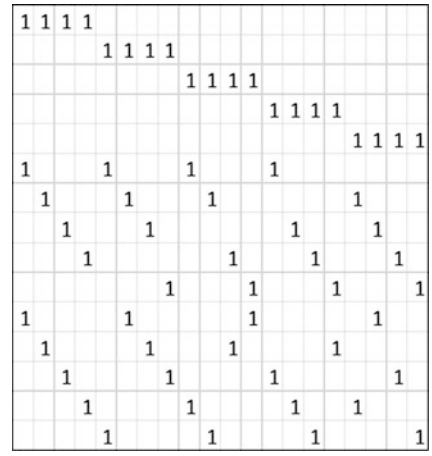


Abb. 8.1 Beispiel einer erweiterten Kontrollmatrix eines LDPC-Codes (auf Basis [Gal])

Da LDPC-Codes nicht gerade effizient konstruier- und implementierbar waren, gerieten sie zunächst in Vergessenheit, bevor sie Ende der 1990er-Jahre wiederentdeckt wurden und heutzutage zu den besten Codes gehören. LDPC-Codes werden also in der Regel über ihre Kontrollmatrizen festgelegt, so wie wir das ja auch schon bei Hamming-Codes gesehen haben. Ein Vorteil dünn besetzter Kontrollmatrizen ist jedenfalls der, dass für LDPC-Codes die Kontrollgleichungen einfach sind, was meist zu effizienten Decodierverfahren führt. Das wesentliche Konstruktionsprinzip von LDPC-Codes ist aber, dass es eigentlich kein einheitliches gibt. Häufig konstruiert man sie so, dass man zuerst ihr Decodierverhalten untersucht und optimiert. Meist sind dabei auch Pseudozufallsverfahren im Einsatz. Wir wollen hier zwei Ansätze etwas näher beleuchten, einen kombinatorischen Ansatz mittels **Tanner-Graphen** und deren **Expandereigenschaft** und die sog. **Repeat-Accumulate-Codes (RA-Codes)**, in beiden Fällen binäre Codes.

8.1.2 Tanner-Graphen

Wenn man noch mal einen Blick auf Gallagers Beispiel wirft, so wird es einem recht schnell *schummrig vor Augen*. Daher ist eine strukturiertere Beschreibung von LDPC-Matrizen hilfreich. Hierzu verwendet man meist sog. **Tanner-Graphen**. Diese wurden von dem amerikanischen Informatiker **Michael Tanner** im Jahr 1981 ursprünglich zur Konstruktion von längeren Codes auf Basis vorhandener kurzer Codes vorgeschlagen, werden aber heute standardmäßig zur Beschreibung von LDPC-Codes genutzt.

Tanner-Graphen haben als linke Ecken die n **Koordinaten** x_i eines binären Wortes und als rechte Ecken die sog. **Prüfbedingungen** y_j, die sich aus den Zeilen der Kontrollmatrix und damit aus den Kontrollgleichungen ergeben. Die Kontrollmatrix kann auch mittels Zeilenlinearkombination um zusätzliche Prüfbedingungen y_j erweitert werden. Dabei werden x_i und y_j mit einer Kante verbunden, wenn x_i in der Prüfbedingung y_j vorkommt.

8.1.3 Beispiel: Tanner-Graphen

Der kleine binäre Hamming-Code $Ham_2(3)$
Als erstes Beispiel nehmen wir folgende Kontrollmatrix, die auch Gallager als Einführungsbeispiel in seiner Arbeit über LDPC-Codes aufführt, und erstellen dazu einen der möglichen Tanner-Graphen.

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

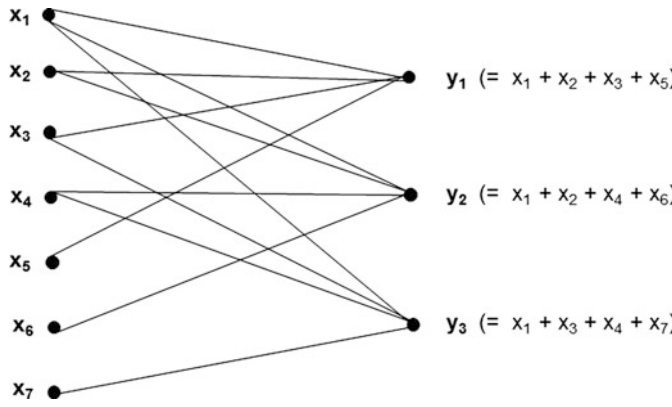


Abb. 8.2 Tanner-Graph zum kleinen binären Hamming-Code

Ein kurzer Blick zeigt, dass dies wieder mal nichts anderes ist als die Kontrollmatrix einer äquivalenten Form des kleinen binären Hamming-Codes $Ham_2(3)$. Hier sind zunächst die Kontrollgleichungen, und zwar nur die für *genau* die Zeilen der Kontrollmatrix:

$$\begin{aligned} x_1 + x_2 + x_3 + x_5 &= 0 \\ x_1 + x_2 + x_4 + x_6 &= 0 \\ x_1 + x_3 + x_4 + x_7 &= 0 \end{aligned}$$

Mit diesen Prüfbedingungen erhält man den in Abb. 8.2 wiedergegebenen Tanner-Graphen. Sind sie erfüllt, d. h. alle $y_j = 0$, so ist $(x_1, \ldots, x_7)$ ein Codewort.

Ein LDPC-Code mit Parameter $[n, k] = [9, 4]$ und mit Expandereigenschaft
Abb. 8.3 zeigt noch ein zweites Beispiel für einen Tanner-Graphen.

Aus den Prüfbedingungen y_j liest man die erweiterte Kontrollmatrix E des zugehörigen LDPC-Codes ab, wobei fünf der sechs Zeilenvektoren von E linear unabhängig sind. Also ist $k = n - 5 = 4$.

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

Was es mit der Expandereigenschaft auf sich hat, erfahren wir jetzt.

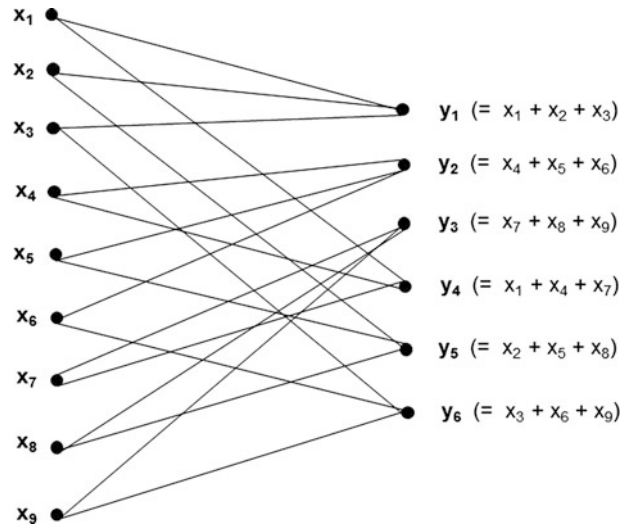


Abb. 8.3 Beispiel eines Tanner-Graphen mit Expandereigenschaft (auf Basis [Wil2])

8.1.4 Expandergraphen

Mit Tanner-Graphen kann man also LDPC-Codes strukturierter beschreiben. Man kann sie aber auch nutzen, um zusätzliche mathematische Methoden aus der **Graphentheorie** einzubringen. Wir wollen hierzu einen kleinen Einblick vermitteln. Zunächst betrachtet man dabei in der Regel nur sog. **reguläre** Tanner-Graphen, bei denen von jeder Koordinate x_i die gleiche Anzahl von Kanten ausgeht und bei denen auch von jeder Prüfbedingung y_j die gleiche Anzahl von Kanten ausgeht. Die eine Anzahl braucht jedoch nicht gleich der anderen zu sein. Für die zugehörige erweiterte Kontrollmatrix heißt das, dass in jeder Zeile und auch in jeder Spalte die gleiche Anzahl von Einsen steht. Dies sind gerade die Typen von LDPC-Codes, auf die man sich laut Gallager besonders konzentrieren sollte. In diesem Sinne ist der Tanner-Graph zu Gallagers Matrix aus Abb. 8.1 regulär sowie auch der Tanner-Graph in Abb. 8.3.

Um Aussagen über den Minimalabstand eines LDPC-Codes ableiten zu können, benötigt man noch eine weitere Bedingung an den zugehörigen Tanner-Graphen, die im Wesentlichen besagt, dass geeignet wenige linke Ecken jeweils mit genügend vielen rechten Ecken durch Kanten verbunden sind, d. h. dass die entsprechenden Koordinaten in hinreichend vielen Prüfbedingungen enthalten sind. Hier ist die exakte, aber etwas technische Formulierung. Ein regulärer Tanner-Graph zu einem LDPC-Code der Länge n heißt **(α, δ)-Expander**, wenn für jede nichtleere Teilmenge X der Koordinatenmenge $\{x_1, \ldots, x_n\}$ mit $|X| \leq \alpha n$ die Ungleichung $|\Delta_X| > \delta |X|$ gilt. Dabei bezeichnet Δ_X die Menge aller Prüfbedingungen y_j, die mit mindestens einer Koordinate aus X verbunden sind. Man überlegt sich leicht, dass der Tanner-Graph in Abb. 8.3 ein $(2/9, \delta)$-Expander ist für jedes $\delta < 3/2$.

Hier ist also die **Expanderschranke** für den Minimalabstand von LDPC-Codes. Sei dazu C ein LDPC-Code der Länge n mit einem (α, δ)-Expander als Tanner-Graph und $\delta \geq s/2$, wobei s die Anzahl der Kanten ist, die von jeder der Koordinaten ausgeht. Dann hat C Minimalabstand $d > \alpha n$.

Die Schranke stammt von **M. Sipser** und **D. Spielman** (1996). Wir nehmen dazu an, dass es ein $0 \neq c = (c_1, \ldots, c_n) \in C$ gibt mit $wt(c) \leq \alpha n$. Sei X die Menge der Koordinaten x_i mit $c_i = 1$. Dann gehen von X insgesamt $s|X|$ Kanten aus. Andererseits enden nach Voraussetzung und wegen der Expanderbedingung diese Kanten in genau $|\Delta_X| > \delta|X| \geq (s/2)|X|$ Prüfbedingungen. Somit gibt es mindestens eine dieser Prüfbedingungen, in der weniger als zwei Kanten – also genau eine – enden. Diese Prüfbedingung hat dann für c den Wert 1, was aber nicht sein kann, da c ein Codewort ist.

Wir kommen im nächsten Abschnitt beim Thema Decodierung auf unseren LDPC-Code mit Expandereigenschaft zurück, wollen uns aber zunächst zwei Praxisanwendungen von LDPC-Codes anschauen.

8.1.5 Anwendung: Festplatten

In Abschn. 5.5 hatten wir uns bereits mit der Datenspeicherung auf CD und DVD auseinandergesetzt. Die größte Bedeutung als Massenspeicher hat jedoch seit mehreren Jahrzehnten das magnetische Speichermedium **Festplatte**. Festplattenlaufwerke sind in Computern verbaut, werden aber auch als externe Laufwerke angeboten. Der Schreib- und gleichzeitig Lesekopf des Schreibfingers ist im Prinzip ein kleiner Elektromagnet, der winzige Bereiche der Scheibenoberfläche unterschiedlich magnetisiert und somit die Daten auf die Festplatte schreibt. Beim Lesen verursachen dabei umgekehrt die Änderungen in der Magnetisierung der Oberfläche durch elektromagnetische Induktion einen Spannungsimpuls im Lesekopf. Festplatten organisieren ihre Daten in sog. Sektoren (mit z. B. 512, 2048 oder 4096 Bytes), die immer nur als Ganzes gelesen oder geschrieben werden können. Auch bei der Datenspeicherung auf Festplatten wird traditionell mit Reed-Solomon-Codes gearbeitet. Da jedoch Festplatten herstellerspezifisch ausgelegt sind, gibt es dort kein standardisiertes Verfahren, wie dies etwa bei CD und DVD der Fall ist. Dennoch sind einige Trends erkennbar, wie hier dargestellt werden soll am Beispiel **HGST** (Hitachi Global Storage Technologies), einem Zusammenschluss von IBM und Hitachi aus dem Jahr 2003, mittlerweile 2012 übernommen von Western Digital.

Als äußerer Code sind der **Reed-Solomon-Code** $\boldsymbol{RS}_{256}(33)$ bzw. geeignete Verkürzungen davon implementiert. Sie werden wie üblich über einem Byte (d. h. 8 Bits) gebildet und können, wie wir wissen, 16 Fehler sicher korrigieren. Als innerer Code setzen sich mehr und mehr **LDPC-Codes** durch. Ihre Konstruktion beruht einerseits auf Eigenschaften des Tanner-Graphen. Die Expandereigenschaft haben wir schon konkret angesprochen. Wichtig ist aber auch, dass der Tanner-Graph keine zu kurzen Zyklen haben darf, d. h. dass man beim Start an einer Variablen mit recht wenigen Schritten innerhalb des Graphen wieder zurück an die Variable gelangen kann. Dies hätte nämlich negative Auswirkungen auf das iterative Decodierverfahren, das wir im nächsten Abschnitt mit einer einfachen Vari-

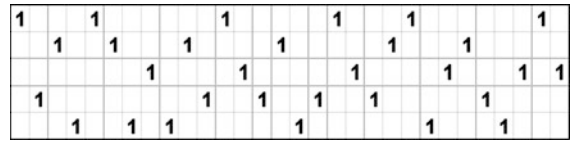


Abb. 8.4 LDPC-Kontrollmatrix für Festplattencodierung (auf Basis [HGST])

ante ansprechen werden. So werden die LDPC-Codes schrittweise mit Pseudozufallsmethoden optimiert. Abb. 8.4 zeigt ein kleines repräsentatives Beispiel einer Kontrollmatrix.

Der zugehörige Tanner-Graph ist regulär, jede Koordinate (Spalte) ist dabei in genau einer Prüfbedingung (Zeile) enthalten, jede Prüfbedingung enthält umgekehrt genau sechs Koordinaten. In der Praxis benutzt man Kontrollmatrizen, bei denen jede Variable in genau zwei, drei bzw. vier Prüfbedingungen enthalten ist. Diese LDPC-Codes sind natürlich länger und haben auch mehr Prüfbedingungen.

Beim Schreiben von Daten auf die Festplatte werden also zunächst auf geeignete Datenpakete der Reed-Solomon-Code angewandt und anschließend die Codewörter des Reed-Solomon-Codes mit dem binären LDPC-Code versehen. Beim Auslesen der Daten korrigiert zunächst der innere LDPC-Code mit seinem schnellen iterativen Decodierverfahren möglichst viele Fehler und dort, wo er versagt, kommt die sichere Korrektur durch den Reed-Solomon-Code zum Einsatz. Bei Festplatten, die ja auf magnetischer Basis funktionieren und zudem in Laufwerken eingebaut sind, spielt das Problem *Kratzer* (*Bursts*) keine so entscheidende Rolle. Daher kommt bei der Codeverkettung in der Regel kein Interleaving zum Einsatz. Stattdessen geht man wie folgt vor. War nämlich ein Sektor schlecht lesbar, d. h. es waren mehrere Leseversuche notwendig bzw. die Fehlerkorrektur zeigte etliche korrigierbare Fehler auf, so wird dessen Inhalt automatisch *geremappt*, d. h. an einer anderen Stelle auf der Festplatte gespeichert. Die möglicherweise beschädigte Stelle wird dann auch zukünftig nicht mehr verwendet.

8.1.6 Anwendung: Satellitennavigation mit GPS (Teil 2)

Wir hatten schon in Abschn. 3.2 über die Codierung der ältesten Technik L1 C/A des **GPS**-Frequenzbands L1 berichtet. Hier wollen wir auf die Nachfolgetechnik L1C zu sprechen kommen. Abb. 8.5 zeigt die Datenstruktur. Jeder Frame besteht dabei aus dem Kopf (TOI) mit 9 Bits, den Uhrzeit- und Positionsangaben mit 576 Bits und weiteren Nutzdaten mit 250 Bits. Die beiden letztgenannten Subframes sind dabei jeweils mit einer 24-Bit-FCS versehen, die mit dem CRC-Polynom CRC-24-Q generiert wird. Wie wir aus Abschn. 6.5 wissen, kann CRC-24-Q Fehler an bis zu 3 Bits erkennen, außerdem jede ungerade Anzahl von Fehlern und Bursts bis zur Länge 24.

Der Kopf wird anschließend mit einem erweiterten binären BCH-Code mit Parametern $[52, 9, 20]$ codiert. Ohne Erweiterung hat der BCH-Code also die Länge 51. Wegen $5 \cdot 51 = 255 = 2^8 - 1$ ist er mittels einer primitiven 51. Einheitswurzel im Körper mit 2^8 Elementen erzeugt und daher *nicht* primitiv.

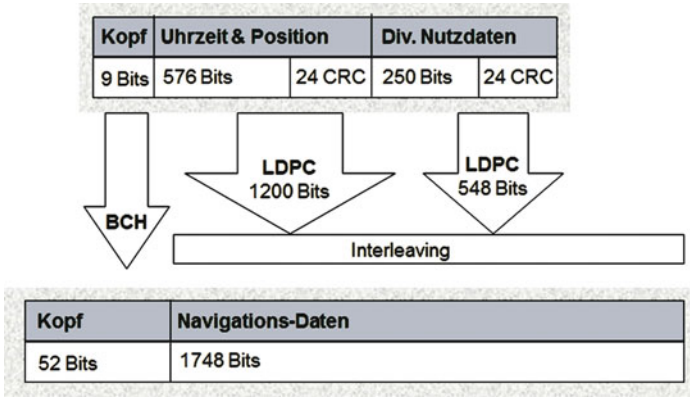


Abb. 8.5 Datenstruktur von GPS L1C (auf Basis [Gar])

Auf die beiden anderen Subframes werden LDPC-Codes mit Parametern $[1200, 600]$ bzw. $[548, 274]$ angewandt und diese anschließend gemeinsam einem Block-Interleaving unterzogen. Dabei ergeben sich insgesamt 52 plus 1748 Bits, die dann gesendet werden. Die LDPC-Codes wurden dabei mit Pseudozufallsverfahren entwickelt, und zwar derart, dass das iterative Decodierverfahren möglichst geringe Decodierfehlerwahrscheinlichkeit besitzt. Nach Decodierung der LDPC-Codes wird mit CRC-24-Q auf Restfehler überprüft und dann ggf. interpoliert oder ignoriert. Bei den Kopfdaten hingegen erfolgt die Fehlerkorrektur mittels BCH-Code. Wir setzten das Thema GPS in Abschn. 9.6 fort.

8.2 RA-Codes und Bit-Flipping-Decodierung

8.2.1 RA-Codes – Repeat and Accumulate

Wie angekündigt kommen wir nun zu einer Teilfamilie der LDPC-Codes, nämlich den **RA-Codes**. RA steht für **Repeat** and **Accumulate** (Wiederholen und Stapeln). Diese Codes wurden 1998 vom amerikanischen Informatiker **Dariush Divsalar** am NASA Jet Propulsion Laboratory unter dem Schlagwort *Turbo-Like Codes* [DJE] vorgeschlagen. Man kann sie nämlich einerseits zu den sog. **Turbocodes** rechnen, die wir aber erst in Kap. 10 kennenlernen werden. Andererseits sind ihre Kontrollmatrizen dünn besetzt, sodass sie meist zu den LDPC-Codes gezählt werden. Das Prinzip ist recht einfach, besteht aber aus einer Codeverkettung, wie wir sie z. B. in Abschn. 5.4 beim Cross-Interleaving (z. B. CIRC) schon untersucht haben und in anderem Zusammenhang auch in Abschn. 9.6 nochmals sehen werden.

Zur Bildung von RA-Codes wird zunächst ein binäres k-Tupel $(x_1, \ldots, x_k)$ aus Informationsbits $r$mal wiederholt (für einen Parameter r). Die zugehörige Generatormatrix G_1

besteht dann aus r Wiederholungen der Einheitsmatrix E_k, d. h. $G_1 = (E_k|\ldots|E_k)$. Der Code ist nichts anderes als ein klassischer Wiederholungscode der Länge $m = rk$, der Dimension k und vom Minimalabstand r.

Danach werden diese m Positionen mittels einer (geeignet wählbaren) Permutation π vertauscht. Die zugehörige Generatormatrix G_2 ist eine sog. *Permutationsmatrix* mit genau einer 1 in jeder der m Zeilen und Spalten. Eine Permutation bedeutet jedoch nur, auf einen äquivalenten Code überzugehen, und ändert daher die Güte des Codes nicht, weder die Dimension noch den Minimalabstand. Wenn man die Generatormatrix $G_1 \circ G_2$ für die Codeverkettung aus beiden Schritten aufschreibt, so ist dies eine Matrix mit k Zeilen und m Spalten, bei der in jeder Spalte genau eine 1 steht und in jeder Zeile genau r Stück.

Nachdem wir bislang nur wiederholt und permutiert haben, sollten wir jetzt auch *stapeln*. Dabei wird ein Wort $(y_1, \ldots, y_m)$ mittels $z_1 = y_1, z_2 = y_1 + y_2, z_3 = y_1 + y_2 + y_3, \ldots, z_m = y_1 + \ldots + y_m$ in das Wort $(z_1, \ldots, z_m)$ überführt. Es wird also ein y_i nach dem anderen *aufgestapelt* und das m-Tupel $(y_1, \ldots, y_m)$ wird dabei in das m-Tupel $(z_1, \ldots, z_m) = y_1(1, \ldots, 1) + y_2(0, 1, \ldots, 1) + \ldots + y_m(0, \ldots, 0, 1)$ codiert. Dieser Code allein bringt natürlich auch nichts, da ja in Wirklichkeit gar keine Redundanz hinzugefügt wird. Dennoch sieht seine Generatormatrix G_3 mit m Spalten und Zeilen wie folgt aus:

$$G_3 = \begin{pmatrix} 1 & 1 & 1 & \ldots & 1 \\ 0 & 1 & 1 & \ldots & 1 \\ 0 & 0 & 1 & \ldots & 1 \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \ldots & 1 \end{pmatrix}.$$

Nun möchte man natürlich noch in systematischer Form codieren. Daher wählt man schließlich als Generatormatrix des RA-Codes $G = (E_k|P)$ mit k Zeilen und $n = k + m = k(1 + r)$ Spalten. Dabei steht die Matrix $P = G_1 \circ G_2 \circ G_3$ für die Verkettung der oben beschriebenen Codiervorgänge und die Matrix E_k stellt das k-Tupel der Informationsbits voran.

Die Konstruktion von **RA-Codes** besteht also aus einer Verkettung von

- Wiederholen,
- Permutieren,
- Stapeln und
- systematischem Codieren,

wie sie in Abb. 8.6 schematisch dargestellt ist.

Abb. 8.6 Schematische Darstellung der RA-Codierung

Warum ist aber die zugehörige Kontrollmatrix H dünn besetzt, sodass man RA-Codes zu den LDPC-Codes zählen kann? Die Kontrollmatrix $H = (H_1|H_2)$ unseres RA-Codes besteht nämlich einerseits aus der Matrix H_1 mit $m = n-k$ Zeilen und k Spalten, welche sich als an der Diagonalen gespiegelte (man sagt auch *transponierte*) Matrix zu $G_1 \circ G_2$ ergibt. Die zweite Matrix H_2 ist folgende Matrix mit $m = n - k$ Zeilen und Spalten:

$$H_2 = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ 1 & 1 & 0 & \dots & 0 \\ 0 & 1 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \dots & 1 & 1 \end{pmatrix}.$$

Um dies allgemein einzusehen, benötigt man etwas Matrizenrechnung. Es ist nämlich $H_2 \circ G_3^t = E_k$ *und folglich* $H \circ G^t = H_1 \circ E_k + H_2 \circ P^t = H_1 + H_2 \circ G_3^t \circ (G_1 \circ G_2)^t = H_1 + E_k \circ H_1 = 2 \cdot H_1 = 0$.

In jedem Fall sollte man sich den Sachverhalt aber anhand des nachfolgenden kleinen Beispiels klarmachen. Immerhin kann man $H = (H_1|H_2)$ getrost als dünn besetzte Matrix bezeichnen.

8.2.2 Beispiel: RA-Codes

Ein kleiner RA-Code mit Parameter [6, 2]
Sei $k = 2$, wir wiederholen zweifach (also $r = 2$) und vertauschen die Positionen 1 und 2 sowie 3 und 4. Dann ist $m = 4$ und

$$G_1 = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}, \quad G_2 = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}, \quad G_3 = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Die Matrix $G_1 \circ G_2$, die zuerst wiederholt und dann permutiert, ist

$$G_1 \circ G_2 = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$

und beim *Stapeln* werden die Zeilenvektoren von $G_1 \circ G_2$ abgebildet gemäß $(0,1,0,1) \to (0,1,1,0)$ und $(1,0,1,0) \to (1,1,0,0)$, also

$$G_1 \circ G_2 \circ G_3 = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix} \text{ und insgesamt } G = \left(\begin{array}{cc|cccc} 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \end{array}\right).$$

Wie man leicht nachrechnet, sieht die Kontrollmatrix dann wie folgt aus:

$$H = \left(\begin{array}{cc|cccc} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \end{array}\right).$$

8.2.3 IRA-Codes

RA-Codes sind einfach strukturiert und ergeben insgesamt eine gute *Performance*. Man kann RA-Codes noch dadurch verbessern, indem man als Verallgemeinerung sog. **IRA-Codes (Irregular Repeat Accumulate)** betrachtet, bei denen dann z. B. nicht jede Position eines k-Tupels $(x_1, \ldots, x_k)$ gleich häufig wiederholt wird bzw. bei denen die Zeilenvektoren der Generatormatrix G_1 des Wiederholungscodes unterschiedlichen Permutationen unterworfen werden. Wir wollen dies nicht weiter systematisch verfolgen, aber stattdessen im nächsten Abschnitt ein sehr komplexes Beispiel eines IRA-Codes kennenlernen. Wie oben angedeutet, sind LDPC-Codes vom IRA-Typ eine Art *Zwitter* zwischen Blockcodes einerseits und Faltungs-/Turbocodes andererseits, wie wir in Abschn. 10.1 noch genauer sehen werden. Bei Letzteren wird der fundamentale Begriff des Minimalabstands so keinen Sinn mehr machen. Stattdessen benutzt man dort **Performancediagramme** bezogen auf diverse Decodieralgorithmen, deren Verwendung sich auch bei LDPC-Codes vom IRA-Typ eingebürgert hat. Dies ist auch der Grund, warum wir hier nicht auf den Minimalabstand eingegangen sind. Ein Beispiel für so ein Performancediagramm sehen wir im nächsten Abschnitt.

8.2.4 Bit-Flipping-Decodierung

Nun wollen wir auch endlich den iterativen Decodieralgorithmus für LDPC-Codes kennenlernen, von dem bislang schon mehrfach die Rede war. Er funktioniert für LDPC-Codes häufig sehr gut. Man benutzt ihn daher gerne im Sinne von *apply the rules and hope for the best*. Für gewisse LDPC-Codes mit Expandereigenschaft kann man auch beweisen, dass er innerhalb der Fehlerkorrekturkapazität immer zum Ziel führt. Wir werden das jedoch nicht tun.

Sei also C ein LDPC-Code der Länge n mit zugehörigem Tanner-Graphen. Die **Bit-Flipping-Decodierung** ist iterativ und geht wie folgt. Sei dazu $v = (v_1, \ldots, v_n) \in \mathbb{Z}_2^n$ das empfangene Wort.

- Initialisiere $u = v$.
- Berechne die Werte aller Prüfbedingungen y_j für u.
- Identifiziere die Menge *korrekter* Prüfbedingungen (d. h. Wert $= 0$) und *inkorrekter* Prüfbedingungen (d. h. Wert $= 1$).
- Entscheide, ob es eine Koordinate x_i gibt, die in mehr inkorrekten als korrekten Prüfbedingungen vorkommt. Falls ja, gehe weiter. Falls nein, gehe zum zweitletzten Schritt.
- Sei $e_i = (0, \ldots, 0, 1, 0, \ldots, 0)$ der Vektor mit 1 an der Stelle i. Setze dann $u \to u + e_i$ (d. h. *flippe das Bit in u an der i. Stelle*) und gehe zurück zum zweiten Schritt.
- Gibt es nur noch korrekte Prüfbedingungen, so korrigiere v zu $u \in C$ und gehe zum nächsten empfangenen Wort. Falls nein, gehe weiter.
- Markiere v als *nicht korrigierbar* und gehe zum nächsten empfangenen Wort.

Wegen der Gefahr von Endlosschleifen sollte man die Iteration nach einer festen Anzahl von Schritten abbrechen. Das Verfahren beruht also auf einer **Mehrheitsentscheidung** (wie auch das Majority-Logic-Verfahren aus Abschn. 4.2). Es handelt sich daher um einen Spezialfall weit dezidierterer Verfahren, bei denen man in jedem Iterationsschritt denjenigen Eintrag ändert, von dem man am *stärksten* glaubt, dass er falsch ist.

8.2.5 Beispiel: Bit-Flipping-Decodierung

LDPC-Code mit Parameter [9, 4] und mit Expandereigenschaft
Wir betrachten nochmals den LDPC-Code C zum Tanner-Graphen der Abb. 8.3. Wir wissen bereits, dass der Tanner-Graph ein (α, δ)-Expander ist für $\alpha = 2/9$ und $\delta = 5/4$. Außerdem gehen von jeder Koordinate $s = 2$ Kanten aus, folglich gilt $\delta = 5/4 \geq 1 = s/2$. Aus der Expanderschranke folgt daher für den Minimalabstand d von C die Abschätzung $d > \alpha n = (2/9) \cdot 9 = 2$, also $d \geq 3$ und C ist 1-fehlerkorrigierend.

Wir führen nun die Bit-Flipping-Decodierung vor. Empfangen werde das Wort $v = (1, 1, 0, 1, 1, 1, 0, 1, 1)$. Die Prüfbedingungen ergeben hierfür $y_1 = y_3 = y_4 = y_6 = 0$ und $y_2 = y_5 = 1$. Die Variable x_5 kommt nur in inkorrekten Prüfbedingungen vor, nicht in korrekten. Wir ändern also das Bit an der 5. Stelle im Wort v und erhalten $u = (1, 1, 0, 1, 0, 1, 0, 1, 1)$. Jetzt sind alle Prüfbedingungen korrekt, d. h. $u \in C$ und v wird daher zu u korrigiert.

Kleiner RA-Code mit Parameter [6, 2]
Wir betrachten nun unser obiges Beispiel C eines sehr kleinen RA-Codes. Die Prüfbedingungen im Tanner-Graphen zur Kontrollmatrix lauten hier $y_1 = x_2+x_3$, $y_2 = x_1+x_3+x_4$, $y_3 = x_2+x_4+x_5$ und $y_4 = x_1+x_5+x_6$. Empfangen werde das Wort $v = (1, 1, 0, 1, 1, 0)$. Hierfür ergeben die Prüfbedingungen $y_1 = y_3 = 1$ und $y_2 = y_4 = 0$. Die Variable x_2 kommt nur in den inkorrekten Prüfbedingungen vor. Wir ändern also das 2. Bit in v und erhalten $u = (1, 0, 0, 1, 1, 0)$. Jetzt sind wieder alle Prüfbedingungen korrekt und v wird zu $u \in C$ korrigiert.

8.2.6 Komplexität: Bit-Flipping-Decodierung

Zunächst sieht man an dem Decodierverfahren, warum man bei Festplatten gerne noch einen Reed-Solomon-Code und bei GPS noch ein CRC-Verfahren nachschaltet. Versagt nämlich die Iteration des Bit-Flipping aus irgendeinem Grund, so hat man immer noch eine Chance das Problem auszubügeln.

Sei also C ein LDPC-Code mit Parameter $[n, k]$. Das Auswerten der Prüfbedingungen für u benötigt maximal $(n - k)n$ elementare Operationen. Das Durchsuchen der x_i in den korrekten und inkorrekten Prüfbedingungen benötigt nochmals $n2(n - k)$ Vergleiche und elementare Operationen. Dieser Zyklus wird nach – sagen wir – maximal e Schritten abgebrochen. In Summe ergibt sich eine Komplexität von $\sim 2e(n-k)^2n^2$. Zum Vergleich beträgt die Komplexität der Syndromdecodierung für binäre Codes $\sim (n - k)2^{n-k}$.

8.3 Digitalfernsehen und IRA-Codes

8.3.1 Der DVB-Standard

DVB, **Digital Video Broadcasting** (dt. **Digitalfernsehen**), bezeichnet einen Standard zur digitalen Übertragung von insbesondere Fernsehen, aber auch von Radio und sonstigen Zusatzdiensten (z. B. MHP, Multimedia Home Platform). Durch Datenkompression (z. B. MPEG2) können mehr Programme pro Sendekanalfrequenz übertragen werden als bei herkömmlichen analogen Verfahren. Für unterschiedliche Übertragungswege gibt es verschiedene Teilstandards, die sich u. a. im Modulationsverfahren unterscheiden. Hier sind die wichtigsten.

- **DVB-S** für die Übertragung durch Satelliten
- **DVB-C** für die Übertragung über Kabelnetzte
- **DVB-T** für die Übertragung durch terrestrische Sender

Zu jedem dieser Standards gibt es bereits entsprechende Nachfolgestandards **DVB-S2**, **DVB-C2** und **DVB-T2.**

Trotz digitalisierter Datenübertragung müssen – wie stets in der Funktechnik – die Bits auf eine analoge Trägerwelle *aufgepfropft* werden, d. h. die Welle wird *digital* moduliert. Zu den einfachsten digitalen Modulationsverfahren zählt die digitale **Amplitudenmodulation** oder auch **Amplitude Shift Keying (ASK)** genannt, bei der die Amplitude des Sendesignals in diskreten Schritten in Abhängigkeit von der Sendedatenfolge umgeschaltet wird. Bei nur zwei Sendesymbolen (Bits) werden zwei unterschiedliche Amplitudenwerte gewählt. Entsprechend wird bei **PSK** die **Phase** der Trägerwelle moduliert. Es gibt auch technisch weiterentwickelte Verfahren, auf die wir nicht weiter eingehen wollen (QAM, QPSK, APSK).

Im DVB-Projekt haben sich Programmanbieter, Gerätehersteller, Netzbetreiber, Verbände und Behörden zusammengeschlossen, um das digitale Fernsehen voranzutreiben. US-amerikanische, japanische und koreanische Firmen sind über ihre europäischen Niederlassungen eingebunden. Auch die Europäische Kommission sowie Normungsorganisationen (**ETSI**, **Europäisches Institut für Telekommunikations-Normen**) sind an der Arbeit beteiligt. **DVB-S** und **DVB-C** wurden 1994, **DVB-T** 1997 ratifiziert. DVB hat das analoge Fernsehen mittlerweile mit Ausnahme des Kabels (geplant bis 2018) komplett abgelöst. **DVB-S2** ist eine Weiterentwicklung des DVB-S-Standards, wobei durch Verwendung verbesserter Codierungs-, Modulations- und Fehlerkorrekturverfahren die Datenrate noch erheblich gesteigert werden konnte. DVB-S2 stammt aus dem Jahr 2005 und unter den Bezeichnungen **DVB-T2** bzw. **DVB-C2** wurden im Jahr 2008 bzw. 2010 zwei weitere Nachfolgestandards festgelegt.

8.3.2 Anwendung: Digitalfernsehen DVB der 1. Generation

Bei den *alten* DVB-Standards wird zur Fehlerkorrektur ein **verkürzter Reed-Solomon-Code mit Parameter $[204, 188, 17]_{256}$** eingesetzt, wie wir ihn bereits in Abschn. 5.4 auf Basis von $RS_{256}(17)$ konstruiert haben. Dieser wird aber zusätzlich mit einem **Faltungscode** verkettet. Wir werden deshalb darauf erst in Abschn. 9.6 näher eingehen.

8.3.3 Anwendung: DVB-S2, DVB-C2 und DVB-T2

Im Rahmen des Ausschreibeverfahrens für DVB-S2 erhielt ein Fehlerkorrekturverfahren den Zuschlag, welches auf einer Verkettung eines **LDPC-Codes vom IRA-Typ** (IRA, Irregular Repeat Accumulate) als innerer Code mit einem äußeren **BCH-Code** beruht. Damit hat dieses Verfahren selbst die *hochgelobten* **Turbocodes** (siehe Kap. 10) geschlagen. Die Fehlerkorrekturverfahren für die neuen Standards DVB-S2, DVB-C2 und DVC-T2 sind im Wesentlichen identisch. Wir fokussieren uns hier auf die Spezifikation des zeitlich ersten Standards DVB-S2.

Bevor wir auf das Gesamtverfahren und die konkreten numerischen Parameter schauen, beginnen wir mit dem komplexesten, nämlich der Spezifikation der Kontrollmatrix H

für den LDPC-Code vom IRA-Typ mit Parameter $[n, k]$. Wie wir das aus dem letzten Abschnitt heraus bereits kennen, setzt sich die Kontrollmatrix $H = (H_1|H_2)$ zusammen aus einer Matrix H_1 mit $n-k$ Zeilen und k Spalten sowie der Matrix

$$H_2 = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ 1 & 1 & 0 & \dots & 0 \\ 0 & 1 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \dots\ 1 & 1 \end{pmatrix},$$

die $n-k$ Zeilen und Spalten hat. Es gilt also noch, die Matrix H_1 festzulegen. Wir wählen dazu eine natürliche Zahl s, die sowohl k als auch $n-k$ teilen soll. Dann wird H_1 wiederum in $r = k/s$ Matrizen M_i zerlegt, die jeweils $n-k$ Zeilen und $s = k/r$ Spalten haben, also $H_1 = (M_1|\dots|M_r)$. Jetzt sind noch die Matrizen M_i zu definieren. Nehmen wir dazu mal an, man hätte schon jeweils die erste Spalte jeder der Matrizen M_i festgelegt. Dann soll sich die zweite Spalte aus der ersten Spalte berechnen, indem man die erste Spalte um den Wert $t = (n-k)/s$ nach unten verschiebt und das Stück, das dann unten herausragt, wieder oben ansetzt. Die dritte Spalte ergibt sich dann ebenso aus der zweiten und so weiter für die restlichen Spalten der Matrix M_i. Man beachte, dass t nach Wahl von s eine natürliche Zahl ist. Wie hat man jetzt aber jeweils die erste Spalte der Matrizen M_i festgelegt? Dies geschah in einem Optimierungsverfahren mit einem Pseudozufallsgenerator, aber mit der Nebenbedingung, dass in jeder Zeile der Matrix H_1 die gleiche Anzahl von Einsen steht. Im Sinne von Tanner-Graphen bedeutet dies, dass jede der Prüfbedingungen des LDPC-Codes mit der gleichen Anzahl von Informationskoordinaten verbunden ist. Der LDPC-Code selbst wird dann wieder mit seiner Generatormatrix $G = (E_k|P)$ beschrieben, die eine Codierung in systematischer Form ermöglicht und bei der sich die Matrix P aus der Kontrollmatrix H derart berechnet, dass die Zeilenvektoren von G und H Skalarprodukt $\langle\ ,\ \rangle$ gleich 0 miteinander haben (d. h. $P = H_1^t \circ G_3$ mit G_3 wie im letzten Abschnitt). Bei den Informationsbits handelt es sich somit um genau die ersten k Stück.

Wir kommen nun zu den konkreten Parametern für den LDPC-Code vom IRA-Typ, der bei DVB-S2 Verwendung findet. Ein Datenpaket (*Frame*) besteht beim Standard-DVB-S2 stets aus 64.800 Bits. Es gibt zwar auch eine verkürzte Variante mit 16.200 Bits. Diese kann als Option eingesetzt werden, um bei Übertragungsbedingungen von sehr geringem Durchsatz die auftretenden Verzögerungen zu reduzieren. Wir beschreiben hier aber nur die Parameter zur Fehlerkorrektur bei Standardframes. Es ist nun so, dass sich die 64.800 Bits auf den Frame *nach* der Codierung beziehen, d. h. der eigentliche Anteil an Information ist geringer. Das Vorgehen ist in Abb. 8.7 visualisiert.

- Jeder Frame beginnt mit einen *Header*, nämlich Kopfdaten mit bestimmten Informationen über den Dateninhalt des Frames.
- Danach kommen die Nutzdaten, die die eigentliche Information tragen.

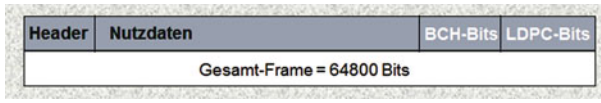


Abb. 8.7 Datenstruktur von DVB-S2

- Diese Sequenz wird zunächst mit einem binären BCH-Code in systematischer Form codiert. Die Paritätsbits dieses äußeren Codes werden der Sequenz angehängt.
- Diese neue Sequenz wird anschließend mit dem oben beschriebenen LDPC-Code in systematischer Form codiert. Auch die Paritätsbits dieses inneren Codes werden der Sequenz angehängt.

Die Gesamtsequenz (*Frame*) muss jetzt 64.800 Bits umfassen, d. h. die Parameter der Codes müssen so gewählt werden, dass am Ende genau 64.800 herauskommt.

Für unterschiedliche Übertragungsbedingungen werden auch beim Standardframe verschiedene Varianten angeboten. Ist die Durchsatzkapazität des jeweiligen Kanals hoch, so kann man mit geringer Coderate, dafür aber mit hoher Fehlerkorrekturkapazität operieren. Bei geringer Durchsatzkapazität sollte man eher höhere Raten verwenden – auf Kosten der Fehlerkorrektur.

Wie sehen nun die verschiedenen Varianten für die LDPC-Codes aus? Es gibt insgesamt elf Varianten mit unterschiedlichen Raten. Zunächst aber vorab:

- Die Spaltenzahl s der r Teilmatrizen M_i, in die man die Matrix H_1 unterteilt, ist stets $s = 360$ und teilt daher alle unten aufgeführten Werte von n und k.
- Die mittels Pseudozufallsgenerator erzeugten ersten Spalten der Matrizen M_i sind je unten aufgeführter Variante in einer großen Adresstabelle des DVB-S2-Regelwerks hinterlegt, worauf wir jedoch nicht weiter eingehen wollen.

Tab. 8.1 zeigt alle zulässigen LDPC-Codes und enthält dabei folgende Angaben:

- Rate des LDPC-Codes $= k/n$,
- $n =$ Anzahl der Codewortbits,
- $k =$ Anzahl der Informationsbits,
- $t =$ Verschiebungswerte für die Spalten der Matrizen M_i gegenüber der 1. Spalte,
- $w =$ Gewicht der Zeilen von $H_1 =$ Anzahl der im Tanner-Graphen verbundenen Informationskoordinaten.

Wer hier in der Tabelle den Minimalabstand und damit die Fehlerkorrekturkapazität im Sinne von Blockcodes erwartet hat, wird enttäuscht. Wie im letzten Abschnitt bereits angesprochen, arbeitet man bei LDPC-Codes vom IRA-Typ mit Performancediagrammen. Abb. 8.8 zeigt ein Beispiel.

Aufgetragen ist dabei die

Tab. 8.1 Zulässige LDPC-Codes für DVB-S2 (Quelle [Fer])

Rate	n	k	t	w
1/4	64.800	16.200	135	4
1/3	64.800	21.600	120	5
2/5	64.800	25.920	108	6
1/2	64.800	32.400	90	7
3/5	64.800	38.880	72	11
2/3	64.800	43.200	60	10
3/4	64.800	48.600	45	14
4/5	64.800	51.840	36	18
5/6	64.800	54.000	30	22
8/9	64.800	57.600	20	27
9/10	64.800	58.320	18	30

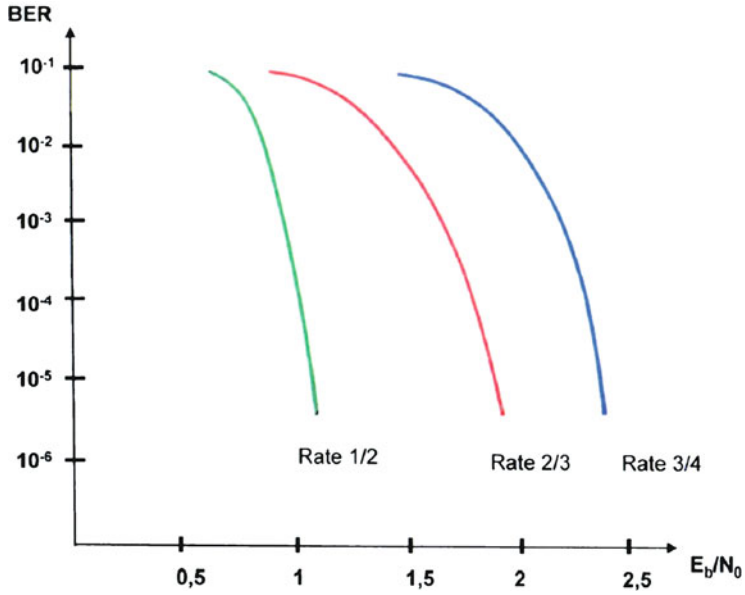


Abb. 8.8 Performancekurven der IRA-Codes für DVB-S2 (auf Basis [Din, VCS])

- **Bitfehlerrate BER** (nach Fehlerkorrektur) über dem
- Verhältnis von **Signalleistung** E_b und **Rauschleistung** N_0 des Kanals
- nach mindestens 20 Iterationsläufen eines Decodieralgorithmus
- für verschiedene Raten des LDPC-Codes.

Wir kommen im Kap. 10 noch genauer auf solche Performancediagramme zu sprechen. Hier nur so viel: Je stärker sich das Signal aus dem Rauschen des Kanals *heraushebt*, umso bessere Fehlerkorrekturwerte sind zu erwarten. Man muss dabei wissen, dass selbst die besten Blockcodes (d. h. Reed-Solomon- und BCH-Codes) günstigstenfalls eine Bitfeh-

Tab. 8.2 Zulässige BCH-Codes für DVB-S2 (Quellen [VCS, ETDVB])

Rate	n'	k'	$n'-k'$	e
1/4	16.200	16.008	192	12
1/3	21.600	21.408	192	12
2/5	25.920	25.728	192	12
1/2	32.400	32.208	192	12
3/5	38.880	38.688	192	12
2/3	43.200	43.040	160	10
3/4	48.600	48.408	192	12
4/5	51.840	51.648	192	12
5/6	54.000	53.840	160	10
8/9	57.600	57.472	128	8
9/10	58.320	58.192	128	8

lerrate von 10^{-6} erreichen können und das nur bei hohem Signal-/Rauschverhältnis, wie Abb. 10.2 zeigt.

Zu jeder der obigen Varianten gibt es nun einen binären BCH-Code als äußeren Code, der die Restfehler noch auszubügeln hilft. Tab. 8.2 listet die zulässigen BCH-Codes auf und enthält dabei folgende Angaben:

- Rate des LDPC-Codes,
- $n' =$ Anzahl der BCH-Codewortbits $= k$,
- $k' =$ Anzahl der BCH-Informationsbits,
- $n' - k' =$ Anzahl der BCH-Paritätsbits,
- $e =$ BCH-Fehlerkorrekturkapazität.

Tab. 8.3 Generatorpolynome der BCH-Codes (Quelle [ETDVB])

Name	Polynom
$g_1(x)$	$1 + x^2 + x^3 + x^5 + x^{16}$
$g_2(x)$	$1 + x + x^4 + x^5 + x^6 + x^8 + x^{16}$
$g_3(x)$	$1 + x^2 + x^3 + x^4 + x^5 + x^7 + x^8 + x^9 + x^{10} + x^{11} + x^{16}$
$g_4(x)$	$1 + x^2 + x^4 + x^6 + x^9 + x^{11} + x^{12} + x^{14} + x^{16}$
$g_5(x)$	$1 + x + x^2 + x^3 + x^5 + x^8 + x^9 + x^{10} + x^{11} + x^{12} + x^{16}$
$g_6(x)$	$1 + x^2 + x^4 + x^5 + x^7 + x^8 + x^9 + x^{10} + x^{12} + x^{13} + x^{14} + x^{15} + x^{16}$
$g_7(x)$	$1 + x^2 + x^5 + x^6 + x^8 + x^9 + x^{10} + x^{11} + x^{13} + x^{15} + x^{16}$
$g_8(x)$	$1 + x + x^2 + x^5 + x^6 + x^8 + x^9 + x^{12} + x^{13} + x^{14} + x^{16}$
$g_9(x)$	$1 + x^5 + x^7 + x^9 + x^{10} + x^{11} + x^{16}$
$g_{10}(x)$	$1 + x + x^2 + x^5 + x^7 + x^8 + x^{10} + x^{12} + x^{13} + x^{14} + x^{16}$
$g_{11}(x)$	$1 + x^2 + x^3 + x^5 + x^9 + x^{11} + x^{12} + x^{13} + x^{16}$
$g_{12}(x)$	$1 + x + x^5 + x^6 + x^7 + x^9 + x^{11} + x^{12} + x^{16}$

Tab. 8.4 Matrixformate für das Interleaving bei DVB-S2 (Quelle [ETDVB])

Zeilen	Spalten
8100	8
5400	12
4050	16
3240	20

Genau genommen handelt es sich dabei jeweils um Verkürzungen von primitiven binären BCH-Codes der Länge $2^{16} - 1 = 65.535$. Durch die systematische Codierung entstehen nämlich nur BCH-Codewörter der angegebenen Längen n', sodass der jeweilige Code um $65.535 - n'$ Bits verkürzt werden kann. Die Generatorpolynome der unverkürzten BCH-Codes ergeben sich über die Fehlerkorrekturkapazität e jeweils als Produkt der ersten e Polynome in Tab. 8.3.

Die LDPC- und BCH-Codes sind direkt verkettet, also ohne zwischengeschaltetes Interleaving. Zur Spreizung von Fehlerbündeln ist jedoch der gesamten Codierung ein Interleaving nachgeschaltet. Abhängig von der Modulation werden nämlich für das Interleaving Matrizen der in Tab. 8.4 aufgelisteten Formate verwendet, die jeweils insgesamt 64.800 Zellen enthalten.

Dabei wird ein Frame mit 64.800 Bits *kunstvoll vertwistet* spaltenweise in die Matrix eingelesen, zeilenweise wieder ausgelesen und dann gesendet. Auf die Art der *Vertwistung* wollen wir nicht weiter eingehen. Jedoch stellt sich die Frage: Was macht es eigentlich für einen Sinn, innerhalb eines Codeworts der immerhin beeindruckenden Länge $n = 64.800$ möglicherweise auftretende Fehlerbündel zu spreizen? Die Antwort lautet: Innerhalb der Welt der Blockcodes *keinen*. Wie wir aber mittlerweile wissen, verwendet man zur Decodierung von IRA-Codes das Konzept des Minimalabstands so nicht. Beim Decodieren wird vielmehr iterativ je Bit über dessen Korrektheit entschieden – wie wir in einem einfachen Beispiel im letzten Abschnitt gesehen haben. Vor diesem Hintergrund macht es dann aber doch Sinn, dass Bitfehler *gespreizt* und somit möglichst breit über die 64.800 Bits verteilt werden.

8.3.4 Anwendung: Drahtloses Funknetz WiMAX

Während das drahtlose lokale Funknetz **WLAN** gemäß Standard IEEE 802.11 für kleinere Reichweiten von wenigen Metern bis mehrere 100 Meter konzipiert ist (siehe Abschn. 6.6), wird **WiMAX** (**Worldwide Interoperability for Microwave Access**) für regionale Funknetze von bis zu 50 km Reichweite eingesetzt. Zur Fehlerkorrektur werden bei WiMAX ebenfalls LDPC-Codes genutzt, die ähnlich wie die vom IRA-Typ konstruiert werden. Sie sind gemäß Standard **IEEE 802.16** für vier verschiedene Raten $n/k = 1/2, 2/3, 3/4$ und $5/6$ vorgegeben, und zwar für die in Tab. 8.5 aufgelisteten Codewortlängen n, aus denen sich dann k entsprechend berechnet.

Tab. 8.5 Längen und Perioden der WiMAX-Codes (Quelle [Fer])

Länge n	Periode z
576	24
672	26
768	32
864	36
960	40
1056	44
1152	48
1248	52
1344	56
1440	69
1536	64
1632	68
1728	72
1824	76
1920	80
2016	84
2112	88
2208	92
2304	96

Die Periode z ist ein Parameter für das Bildungsgesetz der Kontrollmatrizen $H = (H_1|H_2)$. Beide Matrizen H_1 und H_2 werden nämlich wiederum in kleinere Matrizen mit z Zeilen und Spalten unterteilt. Während in H_2 jeweils auf und oberhalb der Diagonalen die Einheitsmatrix E_z steht, besteht die Matrix H_1 neben vielen kleinen 0-Matrizen nur aus kleinen *zyklisch verschobenen* Einheitsmatrizen mit z Zeilen und Spalten, nämlich Matrizen der Form

$$\begin{pmatrix} & 1 & & \\ & \vdots & & \\ 1 & & & 1 \\ & & \vdots & \\ & 1 & & \end{pmatrix}.$$

Die *Verschiebungswerte* sind dabei unterschiedlich. Sie wurden – wie auch die *Verteilungsmuster* der Matrizen – mit Pseudozufallsmethoden optimiert und können im Standard nachgeschlagen werden.

Faltungscodes

9

Faltungscodes – eine andere Welt Nachdem wir es bereits bei IRA-Codes mit einer Art *Zwitter* zu tun hatten, wechseln wir nun ganz in eine andere Welt – die der **Faltungscodes**. Während Blockcodes Informationsblöcke der Länge k in Codeblöcke der Länge n überführen, werden Faltungscodes als eine Codiervorschrift für *quasiunendliche* Bitfolgen aufgefasst, also insbesondere stets binär. Die Codiervorschrift selbst wird durch **nichtrückgekoppelte Schieberegister** beschrieben, bei denen jedes Eingangsbit n Ausgangsbits erzeugt. Man spricht dann von **Rate** $1/n$. Die Idee der Faltungscodes ist so alt wie die Codierungstheorie selbst, nämlich initiiert von **Peter Elias** im Jahr 1955. Das Problem bei Faltungscodes ist aber, dass es keine mit Blockcodes vergleichbare griffige Theorie gibt, sondern Fortschritte eher mit experimentellen, pseudozufälligen Optimierungsmethoden erreicht wurden. Wir befassen uns mit Begriffen wie der **Einflusslänge** und **Gedächtnislänge**, wir lernen, wie man Faltungscodes **oktal klassifiziert** und wie man sie **terminiert**. Außerdem sollten wir uns vor **katastrophalen** Faltungscodes in Acht nehmen.

Trellis-Diagramm und freier Abstand Während die Codiervorschrift bei Faltungscodes einfach – nämlich mittels Schieberegister – beschreibbar ist, ist dies umso weniger der Fall für die erzeugte Codebitsequenz selbst. Wir sehen einen ersten Ansatz aus der Automatentheorie, das sog. **Zustandsdiagramm**. Der zweite Ansatz hat sich aber heute weitgehend durchgesetzt, das sog. **Trellis-Diagramm**, das das Zustandsdiagramm taktweise über die Zeitachse aufrollt, so wie es von **David Forney** im Jahr 1973 vorgeschlagen wurde. Es gibt aber noch einen weiteren Punkt: Das uns lieb gewonnene Gütekriterium Minimalabstand für Blockcodes macht bei quasiunendlichen Bitfolgen so keinen rechten Sinn mehr. Man nutzt stattdessen als *halbherzigen* Ersatz häufig den **freien Abstand**, der – wie wir lernen – nur eine eingeschränkte Aussage über die Fehlerkorrekturkapazität eines Faltungscodes beinhaltet.

Die NASA-Codes – eine kleine (Zeit-)Reise Wir haben schon das ein oder andere Mal einen Blick auf die **NASA-Raumfahrt** und die dabei verwendeten Fehlerkorrekturverfah-

O. Manz, *Fehlerkorrigierende Codes*, DOI 10.1007/978-3-658-14652-8_9

ren geworfen. Nun sind wir an einem Punkt angekommen, an dem wir das Bild weitgehend vervollständigen können. Es begann mit dem **Lin-Lyne-Code** bei **Pioneer 9**, einem komplizierten rekursiven Faltungscode, und weiter über **Mariner** und der **Green-Machine** bis zum **Golay-Code** bei den **Voyager**-Sonden. Wir haben auch schon gehört, dass Voyager 2 für die Reise zum Uranus und Neptun umprogrammiert wurde auf einen **Reed-Solomon-Code**. Dieser wurde aber seinerseits über eine Interleaving-Matrix verkettet mit einem Faltungscode der Rate $1/2$ und Einflusslänge 7, der in der Folge zum **NASA Standard Convolutional Code** avancierte. Das Gesamtkonstrukt hat es sogar zum **NASA Concatenated Planetary Standard Code** gebracht, obwohl man – aus gegebenen Gründen – bei **Galileo** und **Cassini** auf etwas andere Faltungscodes ausgewichen ist. Jedenfalls ist man bei diversen **Mars-Rover**-Missionen dem Standard treu geblieben.

Decodierung mit Viterbi-Algorithmus Wenn man eine Konstante innerhalb der Codierungstheorie ausmachen will, dann ist das die **Viterbi-Decodierung**: Wie bereits 1967 von **Andrew James Viterbi** vorgeschlagen, wurden Faltungscodes und werden sie noch heute im Prinzip nach der gleichen Methode decodiert – und das mit hervorragendem Antwortzeitverhalten. Der Viterbi-Algorithmus ist eine **Maximum-Likelihood-Decodierung**, und zwar bestimmt er die Codebitsequenz mit minimalem Hamming-Abstand zur empfangenen Sequenz. Genauer gesagt macht er das taktweise, indem er sich sukzessive den optimalen Pfad im Trellis-Diagramm sucht. Am besten, wir machen uns das wieder an einem konkreten Beispiel klar.

Punktierte Faltungscodes und Faltungs-Interleaving Wir haben ein Thema bislang offen gelassen, wie man nämlich aus Faltungscodes der Rate $1/n$ solche mit besseren Raten macht. Die Antwort ist verblüffend einfach, man lässt schlichtweg einige Ausgabebits weg. Wenn man beispielweise bei einem Faltungscode der Rate $1/2$ bei jedem zweiten Output ein Ausgabebit weglässt, so hat man daraus einen Faltungscode der Rate $2/3$ gemacht. Man nennt dieses Verfahren **Punktierung**. Kurz beschäftigt haben wir uns auch mit treppenförmig verzögertem Block-Interleaving. Hierfür gibt es eine zweite Implementierung, die wiederum auf **David Forney** zurückgeht. Man nennt sie **Faltungs-Interleaving** und sie basiert auf einer raffinierten Anordnung von Schieberegistern. Das müssen wir uns wieder anhand eines Beispiels verdeutlichen.

Hybridverfahren bei Fernsehen, DSL, Mobilfunk und GPS Nachdem Block- und Faltungscodes nun schon so lange koexistieren, sollte man doch annehmen, dass sie in der Praxis auch gewinnbringend als ***Hybrid*verfahren** eingesetzt werden können. Ja, das stimmt und das haben wir ja auch schon bei den **NASA-Codes** gesehen. Häufig werden dabei – wie auch bei der NASA – Faltungscodes zusätzlich über Interleaving mit einem Blockcode ummantelt. Das hat auch einen Grund. Der Viterbi-Algorithmus macht beim Decodieren des Faltungscodes den ein oder anderen Fehler und macht er einen solchen, so häufen sich diese Fehler aufgrund des Gedächtnisses der Faltungscodes meist zu **Bursts**. Diese erkennt bzw. korrigiert dann der Blockcode, möglicherweise in Kombination mit

Interleaving. Wir schauen uns im Detail noch einige weitere Hybridverfahren an, nämlich das **Digitalfernsehen DVB** der 1. Generation, das schnelle Internet mit **DSL**, den **Mobilfunk** mit den Standards **GSM, UMTS** und **LTE** sowie die Satellitennavigationssysteme **GPS** und **Galileo.**

9.1 Faltungscodes – eine andere Welt

Wir begeben uns nun in eine andere Welt – die der **Faltungscodes**. Hierbei müssen wir einige teils lieb gewonnene Gewohnheiten aus der Welt der **Blockcodes** über Bord werfen.

- Wir werden jetzt nicht mehr den Informationsstring in einzelne Blöcke von k-Tupel aufteilen und diese dann separat als Codewörter in n-Tupel überführen. Stattdessen wird der Informationsstring als kontinuierlich von *quasiunendlicher* Länge aufgefasst und stellen- bzw. bitweise in Strings der Länge n überführt. Dabei ist n relativ klein (im niedrigen einstelligen Bereich).
- Vor diesem Hintergrund macht dann auch der Begriff des Minimalabstands eines Codes so keinen Sinn mehr.
- Ein *ungeschriebenes* Gesetz bei Blockcodes war: *Neues Spiel, neues Glück*. Damit ist gemeint, dass die Codierung eines k-Tupels nicht von in der Folge vorangegangenen k-Tupel abhängt. Auch diese Denkweise müssen wir hinter uns lassen, Faltungscodes besitzen eine Art *inneres Gedächtnis*.

Allerdings muss man auch eines klar vorweg sagen: Für Faltungscodes gibt es keine ausgereifte mathematische Theorie mit algebraischen Konstruktionsprinzipien, wie wir das von linearen Codes her kennen. Vielmehr nutzt man teilweise sehr rechenintensive Simulationen, um Faltungscodes auf ihre *Güte* hin zu überprüfen und ggf. zu optimieren. Aber Faltungscodes sind sehr effizient und schnell implementierbar – nämlich mit Schieberegistern – und sie lassen sich mit dem sog. **Viterbi-Algorithmus** überraschend effektiv decodieren. Das sind die Gründe dafür, dass sie in der Praxis rege Anwendung finden.

9.1.1 Was versteht man unter Faltungscodes?

Faltungscodes (engl. **Convolutional Codes**) leiten sich von dem mathematischen Begriff der *Faltung* ab. Sie wurden erstmals 1955 vorgeschlagen von **Peter Elias** (1923–2001), einem Pionier der Informationstheorie am MIT. Von dem amerikanischen Elektroingenieur und Geschäftsmann **Andrew James Viterbi** (geboren 1935) stammt aus dem Jahr 1967 die Erkenntnis, dass Faltungscodes mit vernünftiger Komplexität und damit Antwortzeit sogar gemäß Maximum-Likelihood decodiert werden können – mit dem berühmten **Viterbi-Algorithmus**.

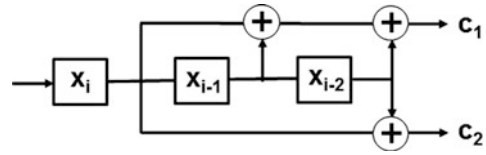


Abb. 9.1 Standardbeispiel eines Faltungscodes

Faltungscodes werden mittels **linearer Gleichungen** definiert und erfordern daher als Alphabet A einen endlichen Körper K. Allerdings werden sie in der Praxis nur binär betrachtet und angewandt, sodass auch wir uns auf den Körper $K = \mathbb{Z}_2$ beschränken. Die Definition von Faltungscodes erfolgt konkret über Schieberegisterschaltungen.

Als Standardbeispiel findet man quasi überall in der Literatur die Schieberegisterschaltung aus Abb. 9.1, die einfach genug ist, um mit ihr zu starten, aber eben auch hinreichend komplex.

Das Verfahren funktioniert so: Bei Beginn der Codierung steht in der linken Zelle das erste Inputbit, die restlichen Speicherzellen (Zustände) werden mit 0 initialisiert. Von links werden dann die Informationsbits der Reihe nach in das Schieberegister geschoben. Das Bit x_i in der linken Zelle ist stets das, welches es zu codieren gilt. Das Register berechnet dann die zugehörigen Codebits, nämlich

- $c_1 = x_i + x_{i-1} + x_{i-2}$,
- $c_2 = x_i + x_{i-2}$.

Diese werden rechts als Output herausgeschoben und mit einem Multiplexer in einen Codebitstring abwechselnd verschachtelt, wie Abb. 9.2 zeigt.

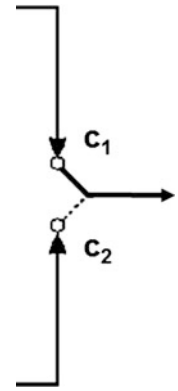


Abb. 9.2 Output-Multiplexer

Man sieht also, dass die Codierung von x_i – außer von x_i selbst – auch von den Vorgängerbits x_{i-1} und x_{i-2} abhängt. In der Literatur wird teilweise die Inputbitzelle x_i nicht als Teil des Schieberegisters aufgefasst, wir werden das aber bei Faltungscodes stets tun.

Hier jetzt gleich zu einigen wichtigen Begriffen bei Faltungscodes. Die **Rate** eines Faltungscodes ist das Verhältnis von Informationsbits k $(= 1)$ zu zugehörigen Codebits n. In unserem Beispiel ist also die Rate $k/n = 1/2$. Der Begriff der Rate entspricht inhaltlich der bei Blockcodes. Man kann grundsätzlich auch Faltungscodes definieren, bei denen $k > 1$ Informationsbits gleichzeitig mittels Schieberegister codiert werden. Wir verzichten aber hier darauf.

Unter der **Einflusslänge** L (engl. **Constraint Length**) eines *nichtrekursiven* (d. h. bei der Schieberegisterschaltung *nichtrückgekoppelten*) Faltungscodes versteht man die Anzahl der Speicherzellen im Schieberegister. Diese beinhalten genau die Informationsbits $x_i, x_{i-1}, \ldots, x_{i-L+1}$, von denen das Codewort zu x_i abhängig ist. Man bezeichnet weiterhin mit $m = L - 1$ die **Gedächtnislänge** des Faltungscodes.

Es sind also genau diese m *Gedächtniszellen* $x_{i-1}, \ldots, x_{i-L+1} = x_{i-m}$, die bei Beginn der Codierung mit 0 initialisiert werden. Unser Eingangsbeispiel ist nichtrekursiv und es gilt $L = 3$ und $m = 2$.

9.1.2 Beispiel: Faltungscodes

Ein nichtrekursiver Faltungscode von Rate 1/3

Zunächst in Abb. 9.3 zur Illustration ein etwas komplexeres Beispiel eines Faltungscodes.

Es handelt dabei um einen Faltungscode mit Rate $k/n = 1/3$ und Einflusslänge $L = 3$. In der Praxis verschaltet man aber normalerweise jeden Ausgang direkt mit dem Eingangsbit x_i, was in unserem Beispiel für den mittleren Output nicht der Fall ist. Hier wird nämlich $x_{i-1} + x_{i-2}$ ausgegeben, die Schaltung um eine Stelle nach links gerückt, wäre dies stattdessen $x_i + x_{i-1}$. Insgesamt wird also am mittleren Output dieselbe Codebitfolge ausgegeben, nur um einen Takt verschoben. Das ändert aber die *Güte* der Gesamtcodesequenz nicht. Letzteres gilt im Übrigen auch für die Reihenfolge, in der die

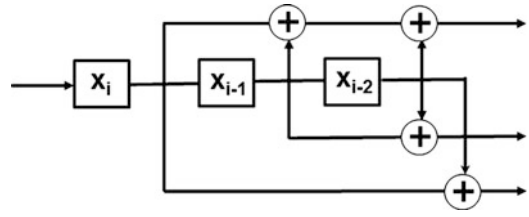


Abb. 9.3 Ein Faltungscode der Rate 1/3

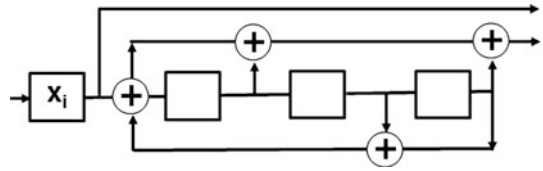


Abb. 9.4 Schaltbild eines RSC-Codes

Schieberegisterschaltungen auf die Ausgänge gelegt werden. Dabei werden jeweils nur die n Codebits – in unserem Beispiel $n = 3$ – in permutierter Reihenfolge ausgegeben.

Die beiden Beispiele legen aber einen Verdacht nahe: Kann man mit unserer Einschränkung $k = 1$ nur Faltungscodes von Rate $k/n = 1/n$ erzeugen? Ja, das stimmt natürlich. Aber um Codes besserer Rate zu erhalten, nutzt man das Mittel der **Punktierung**, auf das wir in Abschn. 9.5 zu sprechen kommen werden.

Ein kleiner RSC-Code

Wir wollen nun anhand Abb. 9.4 einen sog. RSC-Code betrachten.

Dieser Faltungscode ist zunächst einmal in **systematischer** Form, weil die Informationsbits als oberer Output unverändert unter den Codebits erscheinen. Faltungscodes werden aber – im Gegensatz zu Blockcodes – in der Regel nicht in systematischer Form verwendet. Denn dies verringert nur die Rate, bringt aber keinen Vorteil bei der Decodierung, wie wir in Abschn. 9.4 sehen werden.

Dies ist anders bei sog. RSC-Codes. **RSC** steht für **Recursive Systematic Convolutional Code** (dt. **rekursiver systematischer Faltungscode**). Unser Schieberegister erzeugt einen RSC-Code, denn es ist in der unteren Verschaltung **rückgekoppelt** (über eine Rekursionsgleichung). Daher stehen in den $m = 3$ Speicherzellen nicht die Informationsbits, sondern die Werte, die sich über die Rekursionsgleichung aktuell ergeben haben. Leider lässt sich daher die Einflusslänge bei rekursiven Faltungscodes nicht mehr so einfach an der Anzahl der Speicherzellen ablesen, da sie von der Rückkopplungsstruktur abhängt. Um dies einzusehen, wollen wir zu einer Inputbitfolge $x_1x_2x_3x_4x_5\ldots$ die Inhalte der Inputzelle und der drei Gedächtniszellen sowie die beiden Outputcodebits einige Takte lang verfolgen. Wichtig ist dabei ein Blick auf die Outputbits im 7. Takt.

Takt	Inputzelle	Gedächtniszellen			Outputbits	
Init	x_1	0	0	0		
1	x_2	x_1	0	0	x_1	x_1
2	x_3	x_2	x_1	0	x_2	x_1+x_2
3	x_4	x_1+x_3	x_2	x_1	x_3	$x_1+x_2+x_3$
4	x_5	$x_1+x_2+x_4$	x_1+x_3	x_2	x_4	$x_1+x_2+x_3+x_4$
5	x_6	$x_1+x_2+x_3+x_5$	$x_1+x_2+x_4$	x_1+x_3	x_5	$x_2+x_3+x_4+x_5$
6	x_7	$x_2+x_3+x_4+x_6$	$x_1+x_2+x_3+x_5$	$x_1+x_2+x_4$	x_6	$x_3+x_4+x_5+x_6$
7	x_8	$x_3+x_4+x_5+x_7$	$x_2+x_3+x_4+x_6$	$x_1+x_2+x_3+x_5$	x_7	$\boldsymbol{x_1+x_4+x_5+x_6+x_7}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

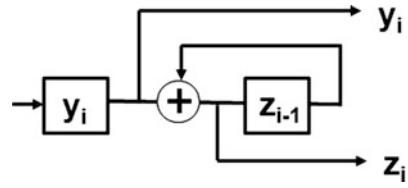


Abb. 9.5 RSC-Code als Basis für RA-Codes

Wir werden jedoch zunächst nur **nichtrekursive Faltungscodes** betrachten, wie dies in den ersten beiden Beispielen der Fall war. Im Zusammenhang mit **Turbocodes** werden aber in Kap. 10 auch RSC-Codes eine Rolle spielen.

Ein RSC-Code zum Stapeln bei RA-Codes

Wir kommen dennoch kurz zurück auf die Konstruktion unseres RA-Codes. Wir haben bei RA-Codes zunächst *wiederholt*, dann *permutiert* und dann *gestapelt*. Hier betrachten wir zunächst nur das *Stapeln*. Dies kann man aber auch so lesen:

$$z_1 = y_1, \quad z_2 = y_1 + y_2 = y_2 + z_1, \quad z_3 = y_1 + y_2 + y_3 = y_3 + z_2, \quad \ldots,$$
$$z_m = y_1 + \ldots + y_m = y_m + z_{m-1}.$$

Es handelt sich daher um eine einstufige Rekursion, allerdings nicht über einen quasi-unendlichen Bitstring, sondern blockweise abgeschnitten nach m Bits. Außerdem haben wir nach dem Stapeln unseren RA-Code noch in systematische Form gebracht, d. h. die Informationsbits mit ausgegeben. Beides zusammen liest sich als Schieberegisterschaltbild eines nach diesen Regeln konzipierten Faltungscodes, wie in Abb. 9.5 dargestellt.

Jetzt beginnen wir zu verstehen, warum RA- und IRA-Codes eine Art Zwitter zwischen Blockcodes einerseits und Faltungs-/Turbocodes andererseits sind. Das Schaltbild zeigt nämlich einen RSC-Code. In Abschn. 10.1 werden wir auf RA-Codes, dann aber als *richtige* Turbocodes zurückkommen.

9.1.3 Oktalkennzeichnung nichtrekursiver Faltungscodes

Wenn wir nun noch mal einen Blick auf unser Eingangsbeispiel werfen, stellen wir natürlich vor dem Hintergrund von Abschn. 6.2 fest, dass die beiden Schieberegisterschaltungen oben und unten jeweils nichts anderes als Polynommultiplikationen ausführen. Hierbei muss man allerdings – da bei Faltungscodes die Bitfolge mit *aufsteigenden* Indizes in das Schieberegister geschoben wird – die Reihenfolge der Polynomkoeffizienten in umgekehrter Reihenfolge lesen. Natürlich möchte man – wenn man einen Faltungscode beschreibt – nicht immer das ganze Schieberegister aufmalen. Kürzer geht die Beschreibung, wenn man nur die zugehörigen Polynome angibt, in unserem Eingangsbeispiel nämlich $g_1(x) = 1 + x + x^2$ und $g_2(x) = 1 + x^2$. Dies wird in der Literatur auch häufig so gemacht, man spricht dann von den **Generatorpolynomen** eines Faltungscodes. Da wir hier aber nicht weiter auf die Polynommultiplikation bei Faltungscodes eingehen,

wollen wir dies nicht weiter vertiefen. Es gibt aber eine andere gängige Kurzbezeichnung, die nichtrekursive Faltungscodes eindeutig festlegt. Man schreibt sich dazu die Schaltungen für alle n Ausgänge binär hin, also 1, wenn die Zelle addiert wird, und 0, wenn sie nicht addiert wird. In unserem Eingangsbeispiel heißt das für oben 111 und für unten 101. Damit ist das Schieberegister eindeutig bestimmt. Damit es einem aber bei großen Schieberegistern nicht *schummrig vor Augen* wird vor lauter Einsen und Nullen, liest man diese Folge **oktal** und schreibt daher 7_8 für 111 und 5_8 für 101 oder kurz $(7_8, 5_8)$. Beim zweiten Beispiel lesen wir am Schieberegister eigentlich 111, 011 und 101 ab. Aber wie dort ausgeführt, verschaltet man standardmäßig immer mit dem Inputbit x_i und damit mit der linken Zelle des Schieberegisters, sodass man diesen Faltungscode als 111, 110, 101 interpretieren würde, also oktal $(7_8, 6_8, 5_8)$. Das üben wir noch an etwas komplizierteren Konstellationen, z. B. $(171_8, 133_8)$. Für mehrstelligen Ziffern wie hier ist es nun so, dass man jede einzelne Ziffer mittels dreier Bits darstellt und links führende Nullen weglässt, da ja immer mit der Eingangszelle des Schieberegisters verschaltet wird. Dies ergibt also $1'111'001$ und $1'011'011$. Ein letztes Beispiel: Der binäre Schaltplan von $(23_8, 35_8)$ ist $10'011$ und $11'101$. Den beiden letztgenannten Faltungscodes werden wir auch später bei Praxisanwendungen wieder begegnen.

9.1.4 Terminierte nichtrekursive Faltungscodes

Eigentlich sollte uns jetzt eines stutzig machen. Mit Polynommultiplikationen hatten wir doch unsere zyklischen Codes codiert. Dann sollten doch eigentlich als Output bei Faltungscodes auch zyklische Codes herauskommen, nämlich Blockcodes ohne Gedächtnis. Wenn man sich aber die Schaltung für zyklische Codes genau in Erinnerung ruft (Abschn. 6.3), so hatten wir nach der Codierung eines k-Tupels die m Gedächtniszellen wieder mit lauter Nullen aufgefüllt, bevor wir zur Codierung des nächsten k-Tupels übergegangen sind, d. h., wir hatten das *Gedächtnis wieder gelöscht*. Dies führt uns zum Begriff der **Terminierung** von Faltungscodes. Bei terminierten Faltungscodes wird nach – sagen wir – je t Informationsbits ein String von m Nullen eingefügt (sog. **Tail-Bits**). Damit sind nach $t + m$ Takten alle m Gedächtniszellen des Schieberegisters wieder mit 0 belegt und das Gedächtnis des Faltungscodes ist wieder gelöscht. Insofern kann man solche terminierten Faltungscodes auch zusammengesetzt aus n Blockcodes auffassen, die Informationstupel der Länge t in Codewörter der Länge $t+m$ abbilden. Damit hat sich die Rate des Faltungscodes insgesamt jedoch auf $t/(t+m) \cdot (1/n)$ verringert. Üblicherweise wird aber ein Faltungscode **terminiert**, d. h., dem Informationsstring werden am Ende jedes Blocks m Tail-Bits angehängt, um wieder einen definierten Zustand zu erzeugen.

9.1.5 Katastrophale Faltungscodes

Zum Ausklang dieses Einführungsabschnitts noch eine Eigenschaft von Faltungscodes, die man aus der Erfahrung von Blockcodes heraus zunächst nicht unbedingt erwarten

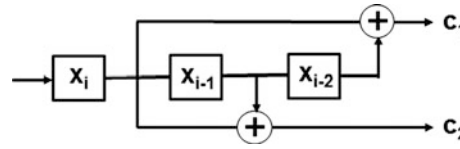


Abb. 9.6 Der katastrophale Faltungscode $(5_8, 6_8)$

würde. Man nennt einen Faltungscode nämlich **katastrophal**, wenn es eine Folge von Informationsbits mit unendlich vielen Einsen gibt, die in eine Codefolge mit endlich vielen Einsen abgebildet wird. Stellen wir uns also mal vor, bei der Datenübertragung würden genau die endlich vielen Einsen in einem solchen Codewort zu 0 verändert. Der Decodierer würde dann diesen 0-String auch als 0-String bei den Informationsbits interpretieren. Die Konsequenz ist: Eine endliche Anzahl vom Fehlern im Übertragungskanal kann zu einer unendlichen Anzahl von Decodierfehlern führen. Katastrophale Codes sind daher zur Datenübertragung ungeeignet und müssen *unbedingt vermieden* werden. Glücklicherweise hat man aber ein einfaches Kriterium für nichtkatastrophale Faltungscodes. Man kann nämlich zeigen, dass ein Faltungscode genau dann nichtkatastrophal ist, wenn die zugehörigen Generatorpolynome $g_i(x)$ teilerfremd sind.

9.1.6 Beispiel: Katastrophale Faltungscodes

Faltungscode $(5_8, 6_8)$
Schieberegister, die katastrophale Faltungscodes erzeugen, sind gar nicht so *exotisch* wie man möglicherweise zunächst glauben könnte. Abb. 9.6 zeigt ein solches Beispiel mit Oktalkennung $(5_8, 6_8)$.

Wir werden uns im nächsten Abschnitt die Codefolge genauer anschauen und uns von der *Katastrophe* dieses Faltungscodes überzeugen. Überlegen wir uns das aber mal anhand des obigen Polynomkriteriums. Die Generatorpolynome sind $g_1(x) = 1 + x^2 = (1 + x)^2$ und $g_2(x) = 1 + x$ und daher nicht teilerfremd, also ist der Faltungscode katastrophal. In unserem Eingangsbeispiel $(7_8, 5_8)$ sind jedoch die beiden Generatorpolynome $g_1(x) = 1 + x + x^2$ und $g_2(x) = 1 + x^2$ teilerfremd, also ist der Faltungscode nichtkatastrophal.

9.2 Trellis-Diagramm und freier Abstand

Bei Blockcodes war die algebraische Beschreibung der Codewörter relativ einfach und übersichtlich, die Herausforderung war dagegen eine effiziente Codierung. Bei Faltungscodes ist das umgekehrt: Die Codiervorschrift ist einfach, wie wir im letzten Abschnitt gesehen haben. Dagegen sind die erzeugten Codebitfolgen in der Regel schwerer zu beschreiben. Zwei der Werkzeuge wollen wir in diesem Abschnitt kennenlernen.

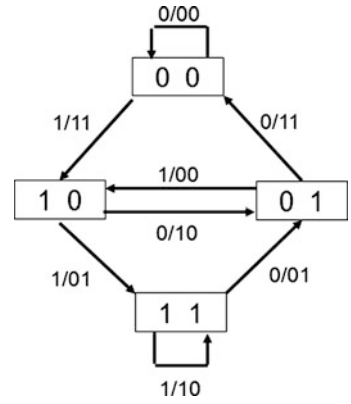


Abb. 9.7 Zustandsdiagramm des Eingangsbeispiels $(7_8, 5_8)$

9.2.1 Zustandsdiagramm

Die Beschreibung eines Faltungscodes mittels **Zustandsdiagramm** stammt aus der Automatentheorie (sog. **Mealy-Automat**).

- Die Kästchen (sog. **Knoten**) des Diagramms beinhalten die Gedächtniszustände des Schieberegisters, also die $m = L - 1$ rechten Zellinhalte.
- Zwei Knoten werden durch einen Pfeil verbunden, wenn der eine Gedächtniszustand des Schieberegisters in den anderen direkt überführbar ist.
- An den Verbindungspfeilen stehen jeweils das Eingabebit, d. h. der Inhalt der linken Zelle im Schieberegisters, sowie die n Ausgabebits.

In Abb. 9.7 ist das Zustandsdiagramm unseres Eingangsbeispiels $(7_8, 5_8)$ dargestellt. Nehmen wir zum Beispiel den Knoten 10 (links im Diagramm). Bei Eingabe 0 wird das zweistellige Codewort 10 ausgegeben und der Gedächtniszustand wechselt zu 01. Bei Eingabe 1 dagegen wird 01 ausgegeben und der Zustand wechselt zu 11. Auf diese Weise bestimmt man die Pfeile zu allen vier Knoten.

Ein Zustandsdiagramm enthält keine Information über den zeitlichen Ablauf der Codierung. Außerdem wird klar, dass bei größerer Gedächtnislänge $m = L - 1$ das Diagramm schnell komplex und unübersichtlich wird.

9.2.2 Beispiel: Zustandsdiagramm

Katastrophaler Faltungscode $(5_8, 6_8)$

Wir üben das Zustandsdiagramm nochmals, und zwar an unserem katastrophalen Faltungscode $(5_8, 6_8)$, und stellen uns einmal vor, dass wir einen Eingabestring aus lauter

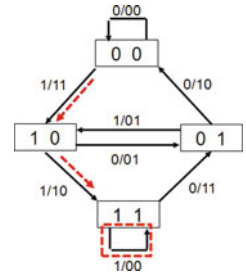


Abb. 9.8 Zustandsdiagramm des katastrophalen Faltungscodes ($5_8, 6_8$)

Einsen verwenden. Abb. 9.8 zeigt das Zustandsdiagramm zusammen mit dem gestrichelten Weg mit lauter Einsen durch das Diagramm. Wegen der Initialisierung der Gedächtniszustände mit 0 bewegen wir uns im ersten Takt vom oberen zum linken Knoten und im zweiten Takt zum unteren Knoten. Hier beginnt nun die *eigentliche Katastrophe*: Wir durchlaufen ab jetzt nur noch die untere Schleife und geben dabei stets 00 aus. Also werden aus den *unendlich* vielen Einsen praktisch nur Nullen (bis auf die ersten zwei Takte).

9.2.3 Trellis-Diagramm

Der wesentliche Nachteil des Zustandsdiagramms ist der fehlende zeitliche Bezug. Dieser zeitliche Bezug kann durch ein **Trellis-Diagramm** (dt. **Netzdiagramm**) visualisiert werden. Ein Trellis-Diagramm rollt sozusagen die Zustandsübergänge über die Zeitachse ab. Ein weiterer Vorteil ist, dass sich im Rahmen des Trellis-Diagramms auch die Viterbi-Decodierung von Faltungscodes anschaulich darstellen und effizient implementieren lässt, wie wir in Abschn. 9.4 sehen werden. Trellis-Diagramme wurden 1973 von **David Forney** vorgeschlagen. Forney war uns bereits im Zusammenhang mit dem sog. Forney-Algorithmus in Abschn. 7.5 begegnet.

Im **Trellis-Diagramm** eines Faltungscodes sind

- vertikal die Gedächtniszustände des Schieberegisters und
- horizontal der Takt des Schieberegisters als Zeitachse

aufgetragen. Zwei Knoten zu aufeinanderfolgenden Takten werden dann verbunden, wenn die Gedächtniszustände ineinander überführbar sind. Die Verbindungslinien werden wiederum mit dem Eingabebit und den Ausgabebits gekennzeichnet.

Ein Trellis-Diagramm beginnt stets im Nullzustand. Dann ist es nach L Schritten voll entwickelt und wird ab dann periodisch fortgesetzt. Abb. 9.9 zeigt das Trellis-Diagramm unseres Eingangsbeispiels ($7_8, 5_8$).

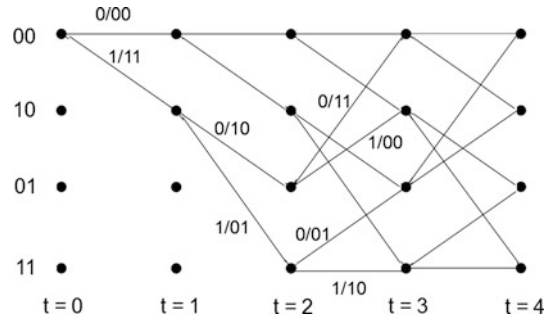


Abb. 9.9 Trellis-Diagramm des Eingangsbeispiels $(7_8, 5_8)$

Wir wollen hier zwei Beobachtungen machen.

- Wenn wir uns eine Eingangsbitfolge vorgeben, etwa 110, so kann man entlang des zugehörigen Pfads im Trellis-Diagramm die Codebitfolge direkt ablesen, nämlich 110101.
- Die Verbindungen zwischen $t = 3$ und $t = 4$ entsprechen genau denjenigen zwischen $t = 2$ und $t = 3$. Da unser Faltungscode Einflusslänge $L = 3$ hat, wird das Trellis-Diagramm ab hier periodisch, d. h., es wiederholt sich bei jedem Schritt.

Wir hatten im letzten Abschnitt darauf hingewiesen, dass man Faltungscodes meist terminiert einsetzt, d. h. am Ende jedes Informationsblocks mit m Stück 0 auffüllt. In der Sprache des zugehörigen Trellis-Diagramms bedeutet das, dass dann sämtliche Pfade wieder oben im Zustand $0 \ldots 0$ enden.

9.2.4 Beispiel: Trellis-Diagramm

Faltungscode $(7_8, 6_8)$

Zur Übung noch ein zweites Beispiel für ein Trellis-Diagramm. Der Faltungscode $(7_8, 6_8)$ liest sich binär als 111 und 110 und hat daher als Schieberegister das in Abb. 9.10 dargestellte Schaltbild.

Die zugehörigen Generatorpolynome $g_1(x) = 1 + x + x^2$ und $g_2(x) = 1 + x$ sind teilerfremd. Daher ist der Faltungscode jedenfalls nichtkatastrophal. In Abb. 9.11 ist das zugehörige Trellis-Diagramm dargestellt.

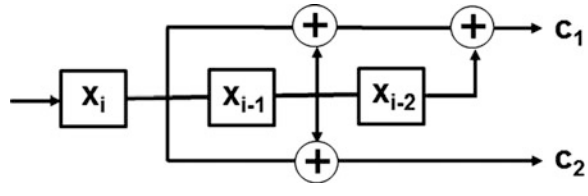


Abb. 9.10 Der nichtkatastrophale Faltungscode $(7_8, 6_8)$

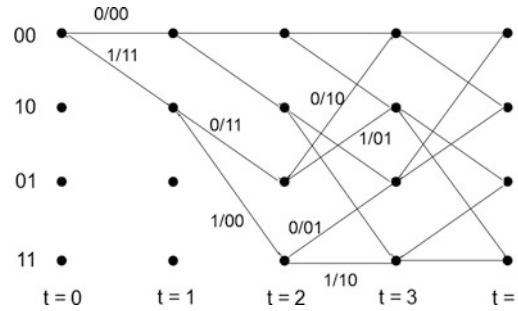


Abb. 9.11 Trellis-Diagramm des Faltungscodes $(7_8, 6_8)$

Schon die Gedächtnislänge $m = 3$ führt zu $2^3 = 8$ Gedächtniszuständen, also zu Trellis-Diagrammen mit acht Zeilen und mehr als $L = m + 1 = 4$ Zeittakten. Wir haben uns daher bei unseren Beispielen auf $m = 2$ beschränkt. Diese Diagramme sind nicht *für Papier* gedacht, sondern als Basis für eine Rechnerimplementierung.

9.2.5 Freier Abstand

Nachdem wir uns nun mit der – leider nicht ganz so transparenten – Darstellung der Codebitfolge eines Faltungscodes auseinandergesetzt haben, kommen wir zurück auf den Begriff des Hamming-Abstands. Sicherlich macht es keinen Sinn, diesen bei Faltungscodes auf die je Takt erzeugten Codeworte der sehr kleinen Länge n zu beziehen, in unseren obigen Beispielen also $n = 2$ oder 3. Wohl macht es aber Sinn, den gesamten Codestring eines Faltungscodes einzubeziehen. Wenn man nämlich weiß, dass je zwei endliche Codesequenzen einer Faltungscodierung **Hamming-Abstand** mindestens d haben und $2e + 1 \leq d$ gilt, so kann man – wie auch bei Blockcodes – bis zu e Fehler korrigieren. Da die Schieberegisterschaltungen lineare Gleichungen für die Codebits liefern, lässt sich – ebenfalls wie bei linearen Blockcodes – dieser Hamming-Abstand d auch einfacher über das **Gewicht** aller endlichen Codesequenzen $\neq 0$ bestimmen.

Wie erhält man aber eine zum **Minimalabstand** bei Blockcodes adäquate Kennzeichnung für Faltungscodes? Klar ist dabei jedenfalls, dass man katastrophale Faltungscodes grundsätzlich bei der Betrachtung ausschließt. Aber über welche Längen von Codebitfolgen soll man dabei das Minimum aller Gewichte beziehen? Sicherlich nicht nur auf die n Outputbits des ersten Takts oder auf einige wenige Takte. Aber ein Faltungscode liefert ja andererseits eine prinzipiell beliebig lange Codebitfolge. Ein Blick auf das Trellis-Diagramm liefert eine Lösung.

Man kann den **freien Abstand** d_f eines nichtkatastrophalen Faltungscodes als das Minimalgewicht aller terminierten Codebitstrings $\neq 0$ definieren, die sich also im

Trellis-Diagramm durch einen Pfad vom Zustand $0\ldots0$ zurück in den Zustand $0\ldots0$ ergeben.

Für unser Eingangsbeispiel $(7_8, 5_8)$ ist der freie Abstand $d_f = 5$, und zwar auf dem kurzen Pfad von 00 nach 00 im Trellis-Diagramm. Beim zweiten Beispiel $(7_8, 6_8)$ ist der freie Abstand $d_f = 4$, diesmal aber nicht auf dem kürzesten Pfad im Trellis-Diagramm.

Der freie Abstand d_f ist grundsätzlich ein **Gütekriterium** für Faltungscodes, wie es der Minimalabstand bei Blockcodes ist. Man muss allerdings auch eines klar sagen: Bei Faltungscodes wird – auch je terminiertem Block – eine Codebitfolge großer Länge produziert, die in der Regel wesentlich größer ist als die für die Definition des freien Abstands notwendigen Pfade im Trellis-Diagramm. Daher bezieht sich der freie Abstand immer nur auf einen relativ kleinen *benachbarten* Bereich in der langen Codebitfolge. In der Realität kann daher ein Faltungscode viel mehr als nur e Fehler korrigieren für $2e + 1 \leq d_f$, wenn diese nicht zu nah beieinander liegen.

Jetzt haben wir einiges an Theorie verinnerlicht und deshalb eine Pause nötig. Lassen wir also deshalb im nächsten Abschnitt die NASA *zu Wort kommen*.

9.3 Die NASA-Codes – eine kleine (Zeit-)Reise

Wir haben in einigen vorangegangenen Abschnitten bereits über den Einsatz von fehlerkorrigierenden Codes bei diversen NASA-Missionen berichtet. Dabei haben wir teilweise um Geduld bitten müssen, da in Kombination mit Blockcodes auch Faltungscodes zum Einsatz kommen. Jetzt sind wir aber gerüstet, die Geschichte weiterzuerzählen. Bevor wir zu unserer Reise starten – und damit wir nicht den Überblick verlieren – als Abb. 9.12 unser Sonnensystem mit seinen acht Planeten Merkur, Venus, Erde, Mars, Jupiter, Saturn, Uranus und Neptun (von innen nach außen).

9.3.1 Anwendung: Pioneer 9

Start 1968 – Sonnenumlaufbahn zwischen Erde und Venus bis 1983
Pioneer 9 war eine Raumsonde der US-Weltraumorganisation NASA im Rahmen des Pioneer-Programms. Sie war die vierte und letzte von insgesamt vier Pioneer-Sonden zur Messung der Sonnenaktivität und deren Auswirkung auf den interplanetaren Raum. Nach dem Start 1968 wurde Pioneer 9 in eine heliozentrische Umlaufbahn zwischen der Erde und der Venus gebracht. Die Sonde war nur auf eine Betriebsdauer von sechs Monaten ausgelegt, lieferte jedoch bis 1983 Daten. Nachdem danach der Kontakt abbrach, wurde die Mission 1987 offiziell beendet. Pioneer 9 war als erste Raumsonde in der NASA-Geschichte mit einem System zur Fehlerkorrektur ausgerüstet. Es handelte sich dabei

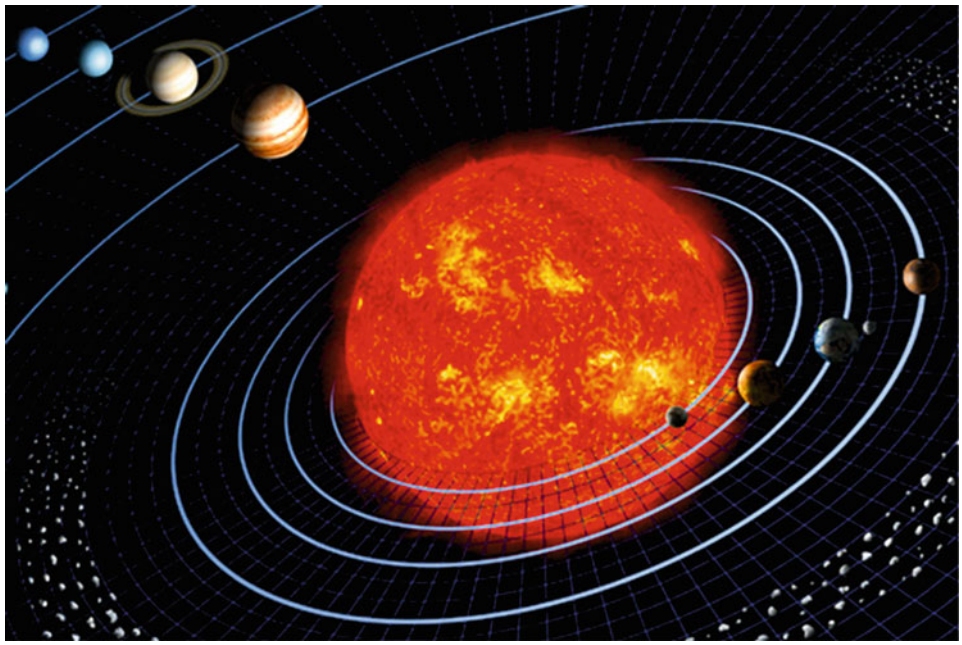


Abb. 9.12 Künstlerische Darstellung unseres Sonnensystems (Quelle NASA [WPPlS])

um den sog. **Lin-Lyne-Code**, einen recht komplizierten, nichtsystematischen, rekursiven Faltungscode von Rate $1/2$, mit $m = 20$ Gedächtniszellen und von Einflusslänge $L = 2(m + 1)$. Decodiert wurde der Code mit dem damals gerade publizierten **sequenziellen Fano-Algorithmus**, einer Variante des Viterbi-Algorithmus, der speziell für Codes mit großer Einflusslänge geeignet ist.

Zur Ursache, warum gerade dieser Code zuerst eingesetzt wurde, lassen wir jetzt **James Massey** zu Wort kommen. In seiner interessanten Retrospektive auf die Anfänge der Codierungstheorie in der Raumfahrt geht er auf den Einfluss des damals verantwortlichen NASA-Mitarbeiters D. R. Lumb ein und sagt dazu Folgendes (Zitat [Mas], S. 16): „After rapid development of the convolutional coding system, Lumb succeeded in getting it aboard Pioneer 9 as an *experiment*. This neatly side-stepped the long approval time that would have been necessary if this coding system had been specified as part of an operational communications system for *a* spacecraft." Und Massey fügt hinzu (Zitat [Mas], S.15f): „The Fano algorithm sequential decoder for the Pioneer 9 system was essentially cost-free – it was realized in software during the spare computation time of an already-on-site computer, whereas the *Green Machine* for decoding the Mariner '69 code was a non-negligible piece of electronic hardware." Allerdings muss man im Nachhinein auch feststellen, dass der Fano-Algorithmus heute meist zugunsten des Viterbi-Algorithmus oder des BCJR-Algorithmus (s. Abschn. 9.4 und 10.2) verschwunden ist.

9.3.2 Anwendung: Mariner 9

Start 1971 – Mars-Umlaufbahn 1972
Nach den Sonden **Mariner 6 und 7** im Jahr 1969 wurde die Sonde **Mariner 9** im Jahr 1971 gestartet und schlug 1972 als erste Sonde in eine Umlaufbahn um den Mars ein. Die Fotos vom Mars wurden mit dem berühmten *Mariner-Code* gesendet, dem binären **Reed-Muller-Code *RM*(1,5)** mit Parameter $[32, 6, 16]_2$. Beim sequenziellen Fano-Algorithmus muss man die Decodierung bisweilen mit Fehler abbrechen, um Langläufer zu vermeiden. Somit hat James Massey den – möglicherweise – entscheidenden Grund angesprochen, warum mit dem Reed-Muller-Code nun ein anderer Code zum Einsatz kam: Die extrem schnelle und zuverlässige Decodierung mit der *Green Machine* (s. Abschn. 4.3).

9.3.3 Anwendung: Voyager 1 und 2

Start 1977 – Jupiter-Passage 1979 und Saturn-Passage 1980/81
Die beiden Sonden **Voyager 1** und **2** wurden beide im Jahr 1977 gestartet, um die Planeten Jupiter und Saturn im Vorbeiflug zu erkunden. Die zwischen 1979 und 1981 von beiden Planeten gesendeten Bilder waren sensationell. Sowohl Voyager 1 als auch die Zwillingssonde 2 waren zur Datenübertragung mit dem binären **Golay-Code** G_{24} mit Parametern $[24, 12, 12]_2$ ausgestattet (s. Abschn. 3.4).

9.3.4 Anwendung: Voyager 2

Start 1977 – Uranus-Passage 1986 und Neptun-Passage 1989
Aufgrund des großen Erfolgs der beiden Missionen wurde entgegen der ursprünglichen Planung entschieden, **Voyager 2** nach der Saturn-Passage in Richtung der Planeten Uranus und Neptun umzulenken. Auf ihrem Weg zu den beiden äußeren Planeten wurden große Teile der Voyager-2-Software remote umprogrammiert, so auch das Fehlerkorrekturverfahren. Dabei wurde G_{24} ersetzt durch den **Reed-Solomon-Code RS_{256}(33)** mit Parameter $[255, 223, 33]_{256}$. Wir hatten in Abschn. 5.3 bereits darauf hingewiesen, dass dieser Reed-Solomon-Code – wie auch schon G_{24} – mit einem Faltungscode verkettet wurde. Da also hierbei zwei Codes aus den beiden unterschiedlichen Welten der Block- und Faltungscodes zum Einsatz kommen, spricht man auch manchmal von ***Hybrid*verfahren**. Beim Faltungscode handelte es sich um den berühmten Code **(171_8,133_8)** mit Einflusslänge $L = 7$ und Rate $= 1/2$, dem **NASA Standard Convolutional Code**. Seine Binärkennung ist $1'111'001$ und $1'011'011$ und ihm zu Ehren sei auch noch sein Schieberegisterschaltbild als Abb. 9.13 wiedergegeben.

Die besten Faltungscodes, d. h. die mit größtem freien Abstand, sind je Rate und Einflusslänge katalogisiert. Als Beispiel findet man in Tab. 9.1 die Faltungscodes der Rate $1/2$ und Einflusslänge bis 10, und zwar mit den Angaben

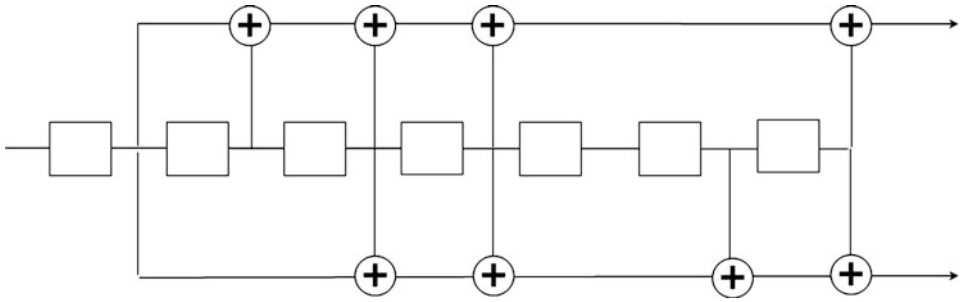


Abb. 9.13 Schieberegisterschaltbild zum Faltungscode $(171_8, 133_8)$

Tab. 9.1 Die besten Faltungscodes der Rate $1/2$ bis Einflusslänge $L = 10$

L	m	Oktalkennung	d_f
3	2	$(7_8, 5_8)$	5
4	3	$(17_8, 15_8)$	6
5	4	$(35_8, 23_8)$	7
6	5	$(75_8, 53_8)$	8
7	6	$(171_8, 133_8)$	10
8	7	$(371_8, 247_8)$	10
9	8	$(753_8, 561_8)$	12
10	9	$(1545_8, 1167_8)$	12

- L Einflusslänge,
- m Gedächtnislänge,
- Oktalkennung,
- d_f freier Abstand.

Hier finden wir natürlich auch den NASA-Code, der folglich freien Abstand $d_f = 10$ hat. Bei $L = 3$ finden wir auch unser Eingangsbeispiel $(7_8, 5_8)$ wieder und werden darin bestätigt, dass der freie Abstand $d_f = 5$ ist. Unser zweites Beispiel für $L = 3$, nämlich $(7_8, 6_8)$, für das wir im letzten Abschnitt auch das Trellis-Diagramm erstellt haben, steht nicht in der Tabelle. Das sollte uns auch nicht wundern, denn wir hatten dafür auch nur $d_f = 4$ berechnet.

Die Codierung bei Voyager 2 erfolgte zuerst mit $RS_{256}(33)$ als äußeren Code und anschließend mit $(171_8, 133_8)$ als inneren Code. Dies ist – nicht ganz überraschend – noch immer nicht die ganze Wahrheit. Zwischengeschaltet wurde zur Korrektur von Fehlerbündeln (Bursts) noch ein **BIock-Interleaver** mit einer **Matrix** von **vier Zeilen** und **255 Spalten**. In diese Matrix wird der Reed-Solomon-Code zeilenweise eingelesen. Die vier Zeilen der Matrix beziehen sich daher auf das Reed-Solomon-Alphabet des Körpers mit $2^8 = 256$ Elementen. Der Faltungscode, der die Matrix dann spaltenweise ausliest, arbeitet jedoch binär, d. h., die Körperelemente müssen dazu als 8-Tupel interpretiert werden. Insofern handelt es sich eigentlich um eine binäre Matrix mit 32 Zeilen und 255 Spalten.

Abb. 9.14 NASA/ESA Concatenated Planetary Standard Code

9.3.5 NASA Concatenated Planetary Standard Code

Auf dieser Basis hat die NASA ihren Standard für fehlerkorrigierende Codes auf interplanetaren Missionen festgelegt, der auch Eingang in den internationalen **CCSDS-Standard** zusammen mit der ESA gefunden hat (siehe Abschn. 10.3). Abb. 9.14 zeigt das gesamte Schema.

9.3.6 Anwendung: Galileo

Start 1989 – Jupiter-Umlaufbahn 1995–2003
Die Sonde **Galileo** sollte ursprünglich bereits 1986 gestartet werden und war mit derselben Konfiguration an fehlerkorrigierenden Codes projektiert wie Voyager 2. Wegen des **Challenger**-Unfalls wurden bei der NASA aber viele Programme verschoben, so auch dieses. Ungünstige Planetenkonstellationen zum geplanten neuen Startzeitpunkt 1989 machten einen Umweg vorbei an Venus und insgesamt eine erheblich längere Reisezeit hin zum Jupiter notwendig. Dies und der kurzfristige Ausfall eines Antennensystems beinhaltete das Risiko von überdurchschnittlich vielen Ausfällen bei der Datenübertragung von Galileo zur Erde. Daher entschloss man sich kurzfristig, die Codekonfiguration zu ändern, und ersetzte den NASA Standard Convolutional Code alternativ durch $(46321_8, 51271_8, 63667_8, 70535_8)$ mit Einflusslänge $L = 15$ und Rate $= 1/4$, den sog. **Galileo Experimental Code**. Dieser Code hat freien Abstand $d_f = 35$. Der Reed-Solomon-Code $RS_{256}(33)$ als äußerer Code blieb unverändert. Ein entsprechender Codierer wurde kurz vor Start in die Sonde eingebaut. Galileo schwenkte 1995 in eine Jupiter-Umlaufbahn ein und stürzte 2003 kontrolliert nach erfolgreicher Mission in den Jupiter.

9.3.7 Anwendung: Cassini/Huygens

Start 1997 – Eintritt Saturn-Umlaufbahn 2004 – geplantes Missionsende 2017, Huygens: Landung auf Titan 2005
Bei **Cassini** handelt es sich um einen Orbiter, der auf einer Umlaufbahn um den Saturn den Planeten selbst und seine Monde untersuchen sollte. **Huygens** wurde als Landungssonde konzipiert, um von Cassini abgekoppelt auf dem Mond Titan aufzusetzen. Die beiden aneinandergekoppelten Sonden wurden 1997 gestartet und 2004 schwenkten beide in die Umlaufbahn um den Saturn ein. Im Jahr 2005 landete Huygens nach

der Trennung von Cassini auf Titan. Das Missionsende für Cassini ist nach mehrfacher Verlängerung aktuell für 2017 geplant. Bei der Cassini-Mission entschloss man sich abermals, vom NASA-Standard abzuweichen. Als Faltungscode verwendete man $(46321_8, 51271_8, 63667_8, 70535_8, 73277_8, 76513_8)$ mit Einflusslänge $L = 15$ und Rate $= 1/6$, den sog. **Cassini-Code**. Dieser Code hat freien Abstand $d_f = 56$. Der Reed-Solomon-Code $RS_{256}(33)$ als äußerer Code blieb unverändert. Es soll hier aber noch einmal klar zum Ausdruck kommen, dass die Auswahl dieser Faltungscodes weniger aus theoretischen Überlegungen heraus getroffen wurde, sondern dass vielmehr langjährige Simulationen für die Entscheidung ausschlaggebend waren. Eigentlich war der Cassini-Code derjenige, den man bereits vor Start von Galileo gefunden hatte. Aber wegen technischer Beschränkungen konnte man dort keinen Code der Rate $1/6$ einsetzten. Deshalb hat man den Galileo Experimental Code von Rate $1/4$ daraus abgeleitet.

Würde man nur den Faltungscode verwenden, so wäre die Bitfehlerwahrscheinlichkeit (BER) viel zu schlecht. Zusammen mit dem Reed-Solomon-Code kommt man aber auf 10^{-6} BER, was für das Cassini-Projekt auch unbedingt notwendig ist. Wenn man dies mit dem Performancediagramm des LDPC-Codes vom IRA-Typ aus Abschn. 8.3 vergleicht, so sieht man, dass dieser etwa in derselben Größenordnung liegt.

9.3.8 Anwendung: Mars-Rover

Pathfinder – Start 1996 – Mars-Landung 1997

Pathfinder wurde 1996 von der NASA gestartet und brachte den ersten Mars-Rover erfolgreich auf die Marsoberfläche. Sie bestand aus einer Landeeinheit mit Kameras und Messinstrumenten sowie einem nur 10 kg schweren Roboterfahrzeug (**Rover**). Die Sonde landete 1997 auf dem Mars in einem Gebiet, wo eine Vielzahl verschiedener Felsen abgelagert ist. Pathfinder und der Rover sendeten mehrere Tausend Bilder sowie mehr als 15 chemische Analysen von Boden und Gestein zur Erde. Noch im selben Jahr fiel die Sonde aus – vermutlich wegen zu kalter Nachttemperaturen. Sie war ursprünglich auch *nur* als Technologiedemonstration für weitere Missionen gedacht.

Spirit und Opportunity – Start 2003 – Mars-Landung 2004 – Betrieb bis 2010 bzw. heute (2016)

Spirit und die baugleiche Schwestersonde **Opportunity** wurden 2003 ebenfalls zur Erforschung des Planeten Mars von der NASA gestartet. Im Gegensatz zu Pathfinder handelte es sich dabei aber um keine feststehenden Bodenstationen, sondern um fahrbare Rover von fast 200 kg Masse. Seit 2010 konnte kein Kontakt mehr zu Spirit aufgenommen werden, was 2011 zu einem faktischen Missionsende führte. Opportunity hingegen ist auch 2016 noch aktiv und hat mittlerweile mehr als 40 km zurückgelegt. Damit fuhr der Rover die weiteste jemals zurückgelegte Strecke auf einem fremden Himmelskörper.

Curiosity – Start 2011 – Mars-Landung 2012 – Betrieb bis heute (2016)

Im Jahr 2011 startete die NASA-Sonde **Curiosity** (auch **Mars Science Laboratory** genannt). Das Kernstück – der **Curiosity-Rover** – wurde im Jahr 2012 mittels einer Abstiegs-

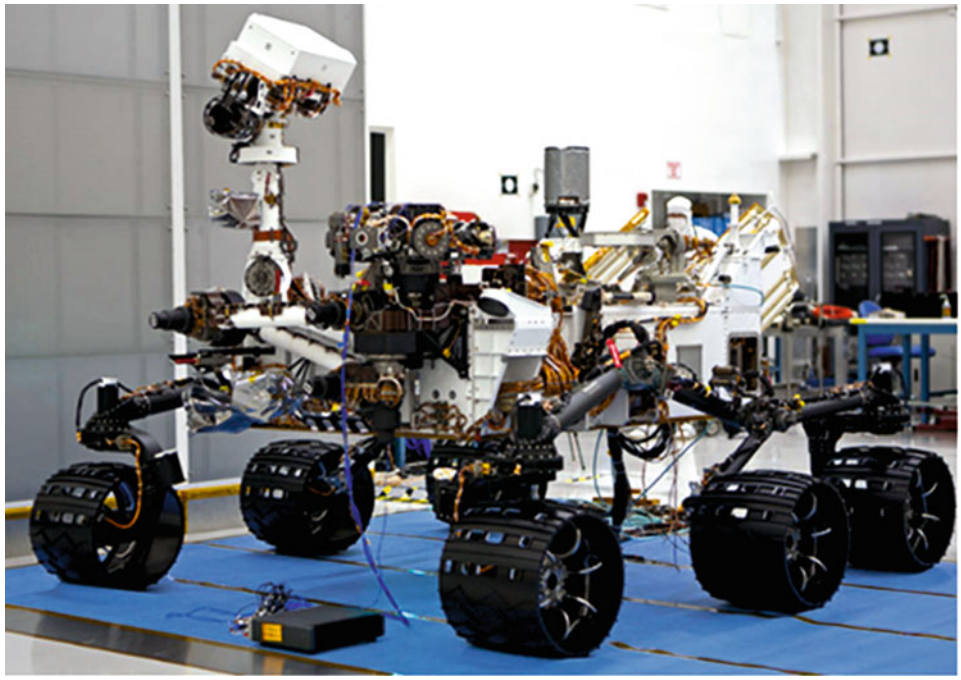


Abb. 9.15 Mars-Rover Curiosity (Quelle NASA [WPMCu])

stufe auf dem Mars gelandet. Er ist mit einer Vielzahl von Instrumenten (Spektrografen, Kameras usw.) zur Untersuchung von Gestein, Atmosphäre und Strahlung ausgerüstet. Mit einer Masse von 900 kg und der Größe eines Pkw ist Curiosity das bislang schwerste von Menschen geschaffene Objekt auf dem Mars. Abb. 9.15 zeigt eine Laboraufnahme.

Wie hat man sich nun aber bzgl. Fehlerkorrekturverfahren bei den Mars-Missionen seitens der NASA entschieden? Trotz des *Hypes* um LDPC-Codes vom IRA-Typ und speziell um Turbocodes verwendete die NASA für die Mars-Missionen weiterhin ihren Standard. Denn außer technischem Fortschritt und möglicherweise besseren Fehlerkorrekturparametern sind auch andere Kriterien zu berücksichtigen, z. B.

- erwiesene Betriebstauglichkeit und Zuverlässigkeit,
- aktuelles Anforderungsprofil,
- Investitionssicherheit über mehrere Jahrzehnte Entwicklung.

Am besten wir hören zum Abschluss dieses Abschnitts ein wenig dem für die Mars-Rover verantwortlichen NASA-Team zu. Jankvist und Toldbod haben im Jahr 2005 das Team besucht, um im direkten Kontakt die Frage nach *Mathematics and People behind the Mission* näher zu beleuchten. Sie kommen dabei zu folgendem Schluss (Zitat [JaTo] S.9): „Mathematics and technology which has been onboard an earlier mission is considered to be safer

and therefore makes *a* more attractive choice. This approach is taken in all aspects of the missions. Of course some development is taking place from mission to mission but only at *a* pace that makes extensive testing possible. ... New ideas which are introduced into the missions will be at least 5–10 years old at launch time, because they must be laid down already when the missions are planned. In the case of channel coding *a* lot of the mathematics involved has to be implemented in hardware for speed. Generally hardware is much more expensive to replace than software, so the gain of introducing *a* new error correcting code has to be considerate in order to balance the expense of the substitution."

9.4 Decodierung mit Viterbi-Algorithmus

Jetzt sollten wir auch endlich erfahren, wie man denn Faltungscodes so effizient decodieren kann und dies auch seit Jahrzehnten praktisch unverändert tut: mit dem **Viterbi-Algorithmus**.

9.4.1 Viterbi-Algorithmus

Beim Viterbi-Algorithmus handelt es sich wieder um eine Maximum-Likelihood-Decodierung.

- Es wird zu einer empfangenen Bitfolge v *die* Codebitsfolge c ausgewählt, für die es am wahrscheinlichsten ist, dass aus ihr die empfangene Bitfolge entstanden ist, das heißt $P(v|c) = \max\{P(v|c')|\, c' \text{ Codebitfolge}\}$.
- Und es ist *die* Codefolge am wahrscheinlichsten, die sich in möglichst wenigen Stellen von der empfangenen Folge unterscheidet, d. h. den geringsten Hamming-Abstand zur empfangenen Folge hat. Das gilt jedenfalls für binäre symmetrische Kanäle, bei denen nämlich die Bitfehlerwahrscheinlichkeit gleich $p < 1/2$ ist.

Mit diesem Ansatz hatten wir bei Blockcodes in Abschn. 2.4 unsere ersten *Gehversuche* zum Thema Decodieren gemacht. Bei nicht allzu großen Blockcodes ist die Suche in der Liste aller Codewörter nach *dem* Codewort, das den geringsten Abstand von einem empfangenen Wort hat (Hamming-Decodierung), praktikabel und bei kleinen Codes manchmal sogar schneller als andere dezidierte Verfahren. Das entsprechende Vorgehen für Faltungscodes könnte nun so sein, dass man zu einer empfangenen Bitfolge aus allen möglichen Codebitfolgen diejenige mit dem geringsten Hamming-Abstand auswählt. Was für Blockcodes noch denkbar war, geht hier faktisch nicht mehr, denn dies wird viel zu aufwendig, da die Anzahl der Folgen exponentiell mit ihrer Länge wächst. Es ist daher in der Praxis nicht realisierbar. Die Idee hinter dem Viterbi-Algorithmus ist einfach die, den Hamming-Abstand sukzessive zu berechnen. Und es war der Vorschlag von Forney, dies am besten innerhalb des Trellis-Diagramms zu tun.

Und so funktioniert der **Viterbi-Algorithmus**.

- Wir setzen voraus, dass unser Faltungscode der Rate $1/n$ mit Einflusslänge L und Gedächtnislänge $m = L-1$ terminiert, d. h. mit m Stück 0 aufgefüllt wurde. Dann enden alle Pfade im zugehörigen Trellis-Diagramm oben rechts im Zustand $0 \ldots 0$.
- Zu jedem Pfad im Trellis-Diagramm gehört außerdem sowohl eine Informationsbitfolge als auch die zugehörige Codebitfolge.
- Wir notieren nun zu jedem Takt t an den zugehörigen Knoten im Trellis-Diagramm den Hamming-Abstand zwischen empfangener Bitfolge bis zum Takt t und den Codebitfolgen zu allen Pfaden, die an diesem Knoten enden.
- Für den nächsten Takt *überleben* an jedem Knoten zu Takt t nur noch *die* Pfade mit Endpunkt an diesem Knoten, die den kürzesten Hamming-Abstand aufweisen. Alle anderen werden aus dem Trellis-Diagramm gelöscht; dabei können ggf. auch beide überleben.
- Dieses Prozedere wird von Takt zu Takt fortgesetzt. Da immer die Hamming-Abstände bei den Vorgängerknoten notiert wurden, muss man also nur noch den Hamming-Abstand für das n-Tupel des letzten Takts berechnen und hinzuaddieren.
- Am Ende des Trellis-Diagramms ergibt sich ein **Survivor**-Pfad. An diesem Pfad kann man die gewünschte Informationsbitfolge direkt ablesen. Der Aufwand für dieses Vorgehen wächst nur noch linear mit der Länge der empfangenen Bitfolge.

Man kann mit etwas Wahrscheinlichkeitsrechnung und einer daraus abgeleiteten *Abstandsmetrik* formal nachweisen, dass der Survivor-Pfad wirklich der gewünschte Pfad für die Maximum-Likelihood-Decodierung ist. Wir wollen das aber hier nicht tun.

9.4.2 Beispiel: Viterbi-Decodierung

Faltungscode ($7_8, 5_8$)
Stattdessen führen wir die Viterbi-Decodierung an einem Beispiel vor, nämlich an unserem Eingangsbeispiel $(7_8, 5_8)$ aus Abschn. 9.1 und 9.2.

Eigentliche Informationsbits:	1	0	1	1		
Tail-Bits zur Terminierung:	0	0				
Informationsbitfolge:	1	0	1	1	0	0
Codebitfolge:	11	10	00	01	01	11
Empfangene Bitfolge (mit 2 Fehlern):	11	10	**1**0	01	0**0**	11

In Abb. 9.16 bis 9.23 wird die Vorgehensweise Schritt für Schritt beschrieben. Die Trellis-Diagramme enthalten zusätzlich die Hamming-Abstände der an den jeweiligen Knoten endenden Pfade zur empfangenen Bitfolge. Und so geht man vor.

- Bestimmung der Hamming-Abstände für die ersten beiden Takte
- Bestimmung der Hamming-Abstände für den 3. Takt
- Auswahl der Pfade mit geringstem Hamming-Abstand nach dem 3. Takt
- Bestimmung der Hamming-Abstände für den 4. Takt
- Auswahl der Pfade mit geringstem Hamming-Abstand nach dem 4. Takt
- Bestimmung der Hamming-Abstände für den 5. Takt unter Berücksichtigung der Terminierung

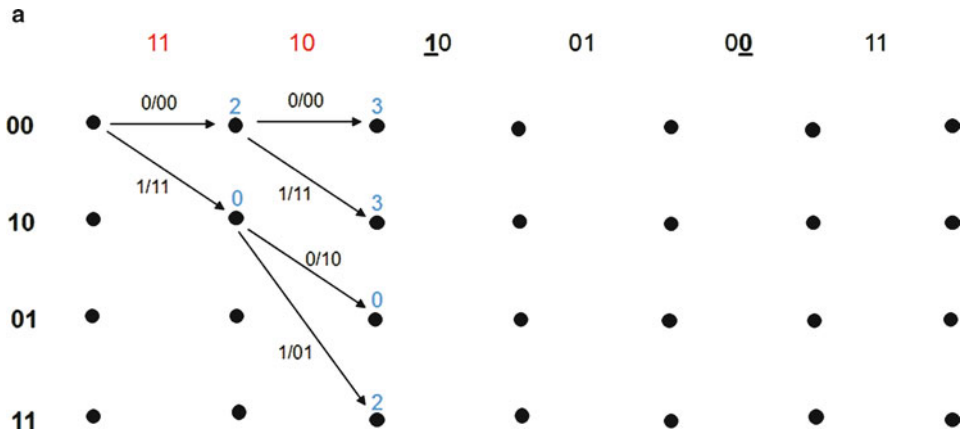


Abb. 9.16 Viterbi-Decodierung – Schritt 1

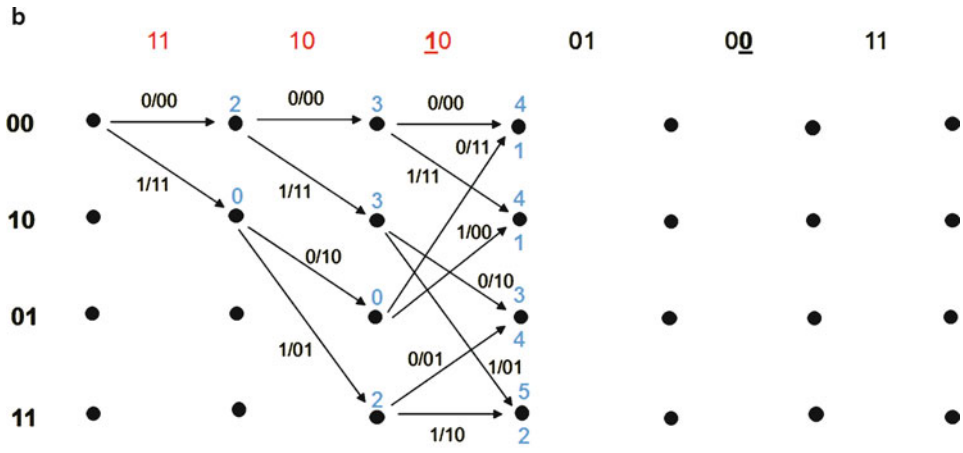


Abb. 9.17 Viterbi-Decodierung – Schritt 2

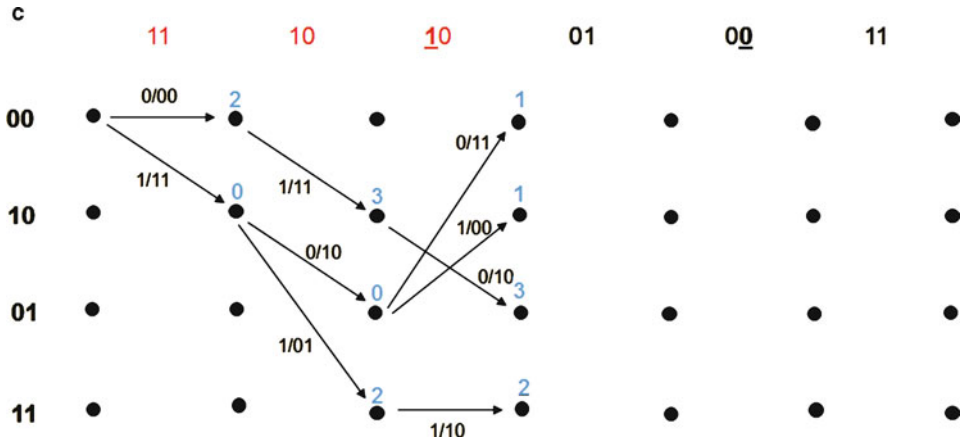


Abb. 9.18 Viterbi-Decodierung – Schritt 3

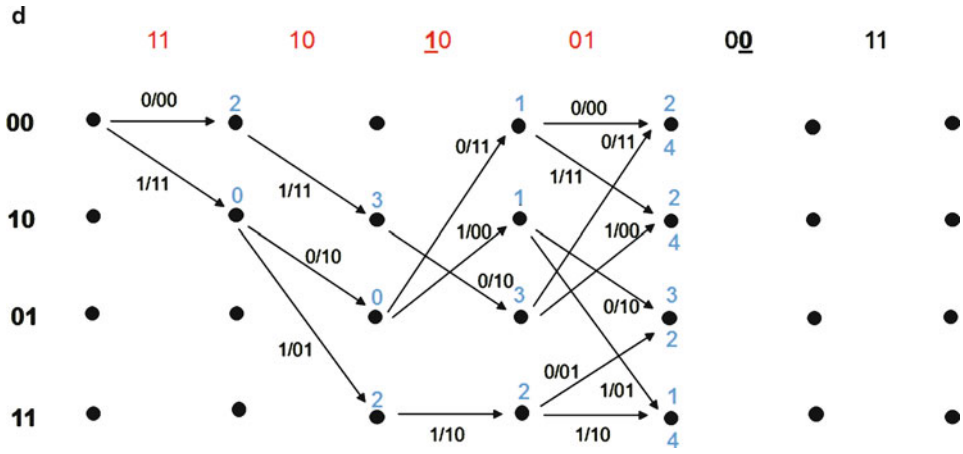


Abb. 9.19 Viterbi-Decodierung – Schritt 4

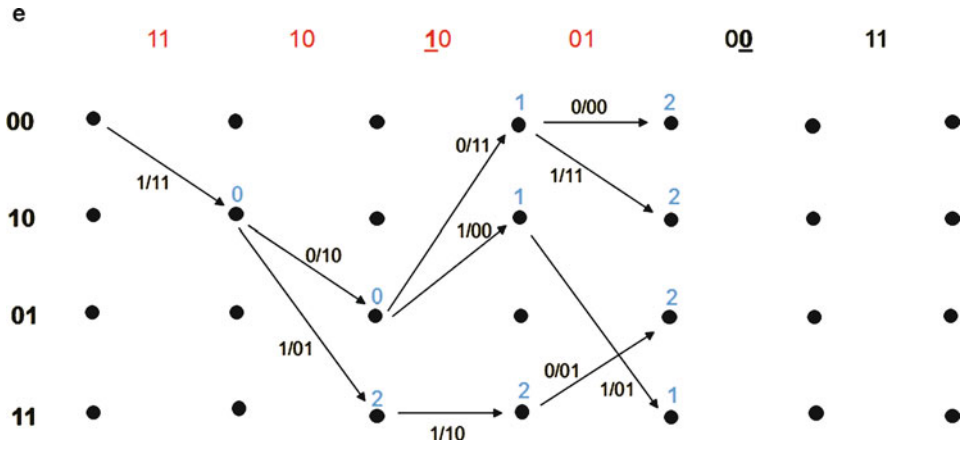


Abb. 9.20 Viterbi-Decodierung – Schritt 5

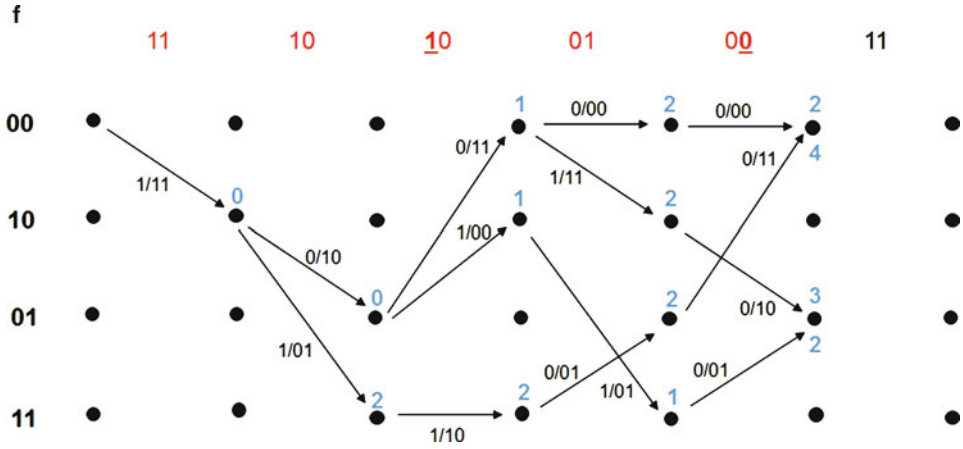


Abb. 9.21 Viterbi-Decodierung – Schritt 6

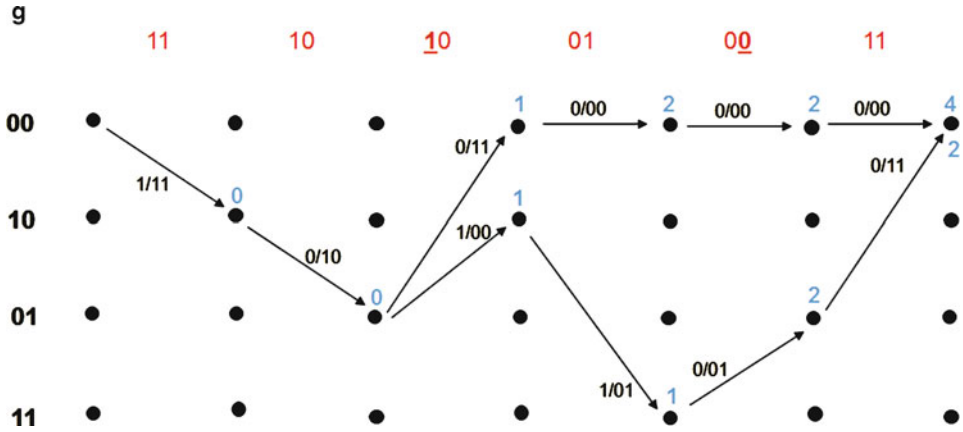


Abb. 9.22 Viterbi-Decodierung – Schritt 7

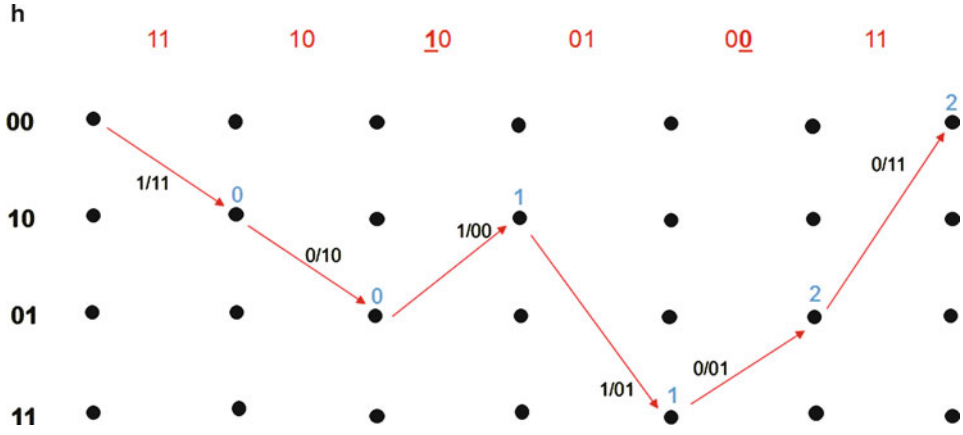


Abb. 9.23 Viterbi-Decodierung – Schritt 8

- Auswahl der Pfade mit geringstem Hamming-Abstand nach dem 5. Takt und Bestimmung der Hamming-Abstände für den 6. Takt unter Berücksichtigung der Terminierung
- Auswahl des Survivor-Pfads

Der Survivor-Pfad hat Hamming-Abstand 2 von der empfangenen Bitfolge. Man kann an ihm auch direkt die Informationsbitfolge ablesen, nämlich 1 0 1 1 0 0.

9.4.3 Fehlerbündel (Bursts) bei Viterbi-Decodierung

Es ist leicht einzusehen, dass der Viterbi-Algorithmus bei nichtkorrekter Decodierung meist ganze Stücke falscher Codestrings liefert. Denn wenn die Entscheidung bei einem Takt mal für den falschen Pfad erfolgte, dann werden aufgrund des *Gedächtnisses* der Faltungscodierung auch die nächsten Takte dadurch beeinflusst. Daher produziert die Viterbi-Decodierung – wenn sie einmal falsch ist – dann meist auch Fehlerbündel. Dies sind also *Bursts*, die diesmal nicht durch den Kanal, sondern durch die Decodierung verursacht sind. Dies macht Faltungscodes als innere Codes bei Codeverkettung oder Interleaving-Verfahren besonders geeignet. Denn dann korrigiert zunächst der Faltungscode als innerer Code mit seiner schnellen Viterbi-Decodierung die meisten Fehler, produziert aber dabei das ein oder andere Fehlerbündel. Diese werden dann von einem äußeren Code – in der Regel Reed-Solomon- oder BCH-Codes – korrigiert. Diese Konstellation haben wir so auch schon im letzten Abschnitt beim NASA Concatenated Planetary Standard Code gesehen.

9.4.4 Rückgrifftiefe und Fast-Synchron-Decodierung

Man stellt beim Viterbi-Algorithmus fest, dass zu jedem Takt t bei Zurückverfolgung der *überlebenden* Pfade ab einem Takt $t-r$ alle den gleichen Ausgangspfad besitzen. Ein solches r nennt man **Rückgrifftiefe** des Faltungscodes. Bei Faltungscodes mit Rate $1/2$ und Gedächtnislänge m liegt r etwa bei $5m$. Bei Takt t ist also bereits über die Decodierung der empfangenen Bitfolge bis Takt $t-r$ entschieden. Somit kann man – zusätzlich zum Fortgang des Viterbi-Algorithmus – schon synchron die Informationsbitfolge zeitversetzt bis Takt $t-r$ ausgeben.

9.4.5 Komplexität: Viterbi-Decodierung

Ein Faltungscode von Rate $1/n$ und Gedächtnislänge m hat insgesamt 2^m Zustände und die Anzahl der elementaren Operationen, die während des Vorwärtsdurchlaufs je Takt für einen Zustand durchgeführt werden sind

- $2n$ Vergleiche und $2n$ Additionen zur Berechnung des Hamming-Abstandes im aktuellen Takt,
- 2 Additionen zur Berechnung des gesamten Hamming-Abstands und 1 Vergleich.

Damit ergeben sich für den Vorwärtsdurchlauf für eine Informationsbitfolge der Länge t genau $2^m(4n+3)t$ elementare Operationen. Beim Rückwärtsdurchlauf wird am Survivor-Pfad jeweils ein Informationsbit je Takt abgelesen. Dies benötigt nochmals t elementare Operationen. Insgesamt hat der Viterbi-Algorithmus – in der vorgestellten Form – eine Komplexität von $\sim (2^m(4n+3)+1)t$.

Insgesamt ist der Viterbi-Algorithmus also linear in der Länge t und hat ein vernünftiges Antwortzeitverhalten für nichtrekursive Faltungscodes mit einer Einflusslänge $L \leq 15$. Dies ist auch der Bereich, für den Faltungscodes in der Praxis eingesetzt werden. Erst darüber hinaus macht der im letzten Abschnitt kurz angesprochene Fano-Algorithmus Sinn. Das Viterbi-Decodiermodul wird üblicherweise in Assembler programmiert und als **ACS-Modul** bezeichnet (**Add-Compare-Select**).

9.5 Punktierte Faltungscodes und Faltungs-Interleaving

9.5.1 Punktierte Faltungscodes

Bei Blockcodes hatten wir ja stets das Ziel, unter Berücksichtigung der benötigten Fehlerkorrekturkapazität Codes mit möglichst großer Rate zu konstruieren, um die Transportkapazität des Kanals optimal auszunutzen. Dies gilt natürlich auch weiterhin bei Faltungscodes.

Um die Rate zu vergrößern, lässt man bei Faltungscodes der Rate $1/n$ einige Codebits bei der Datenübertragung weg. Dieses Verfahren nennt man **Punktierung**. Den Code selbst nennt man **punktierten Faltungscode** (**PCC**, **Punctured Convolutional Code**), den Ausgangscode der Rate $1/n$ **Muttercode**.

Dies macht man natürlich nicht zufällig, sondern systematisch. Man legt dazu einerseits eine **Punktierungsperiode** p fest, bei der sich das Schema wiederholt. Und für die je Takt im Muttercode erzeugten Codewörter der Länge n legt man andererseits ein individuelles Punktierungsschema fest. Dies ergibt eine Matrix P mit n Zeilen und p Spalten, die **Punktierungsmatrix**. Diese legt in jeder Spalte fest, ob ein Bit gesendet ($= 1$) oder ob es unterdrückt ($= 0$) werden soll.

Wir nehmen wieder unser Eingangsbeispiel $(7_8, 5_8)$ von Rate $1/2$ und wählen als Punktierungsmatrix

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

Dabei wird also bei jedem 2. Codewort jeweils das 2. Bit weggelassen. Aus 2 Informationsbits werden jetzt also nicht 4 Codebits, sondern nur noch 3. Dies ergibt folglich einen punktierten Code von Rate $2/3$. Die Punktierungsmatrix

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

erzeugt also aus 3 Informationsbits keine 6 Codebits mehr, sondern nur noch 4. Also ergibt dies einen punktierten Code von Rate $3/4$.

Zum Decodieren muss natürlich der Empfänger die Punktierungsmatrix kennen. Weiterhin ist zu beachten, dass das Trellis-Diagramm eines PCC dasselbe ist wie das seines Muttercodes, wobei aber an den Pfaden als Codebitfolge die punktierten Positionen mit *Platzhaltern* geführt werden. Man kann auch den Viterbi-Algorithmus verwenden und ignoriert dabei einfach die Platzhalterpositionen. Jedoch ist zu beachten, dass sich durch Punktierung der freie Abstand d_f eines Faltungscodes und damit seine Distanzeigenschaften verringern können und sich dabei sogar ein katastrophaler Code ergeben kann. In jedem Fall muss beim Viterbi-Algorithmus die Rückgrifftiefe entsprechend verlängert werden, um weiterhin sichere Entscheidungen zu ermöglichen. Ein wesentlicher Vorteil der Punktierung ist, dass ohne mehrfachen Hardwareaufwand eine flexible Coderate durch geeignete Bitunterdrückung erzeugt werden kann und man damit auf aktuelle Übertragungsbedingungen reagieren kann.

9.5.2 Beispiel: Punktierte Faltungscodes

Faltungscode $(171_8, 133_8)$
Hier sind die besten PCCs zum berühmten $(171_8, 133_8)$-Code mit Rate $= 1/2$ und mit freiem Abstand $d_f = 10$.

Rate $= 2/3$: freier Abstand $d_f = 6$ und Punktierungsmatrix

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

Rate $= 3/4$: freier Abstand $d_f = 5$ und Punktierungsmatrix

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

Rate $= 5/6$: freier Abstand $d_f = 4$ und Punktierungsmatrix

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

Rate $= 7/8$: freier Abstand $d_f = 3$ und Punktierungsmatrix

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

9.5.3 Faltungs-Interleaving

Das **Faltungs-Interleaving** entspricht prinzipiell dem treppenförmig verzögerten Block-Interleaving (siehe Absch. 5.4). Hierbei werden die Zeichen jedoch mit einer Schieberegisterschaltung innerhalb eines Datenstroms gespreizt.

Es geht zurück auf **D. Forney** (1971) und unabhängig davon auf **J. Ramsey** (1970). Die Funktionsweise eines Faltungs-Interleavers wird nun anhand von Abb. 9.24 erläutert.

- Es sei n die Länge eines Datenpakets. In einigen Anwendungen – aber nicht allen – ist dies auch gleichzeitig die Länge eines äußeren Blockcodes. Sei weiter m ein Teiler von n und z eine natürliche Zahl mit $z \geq n/m$. Man bezeichnet dann m als die **Interleaving-Tiefe** und z als die **Basisverzögerung**.
- Der **Interleaver** besteht aus insgesamt m Verschaltungen, wobei eine direkt ist und die anderen jeweils aus einem Schieberegister mit $z, 2z, \ldots, (m-1)z$ Zellen bestehen. Zwei Schalter – ein Multiplexer und ein Demultiplexer – verbinden jeweils die Schaltungen mit dem Ein- bzw. Ausgang.

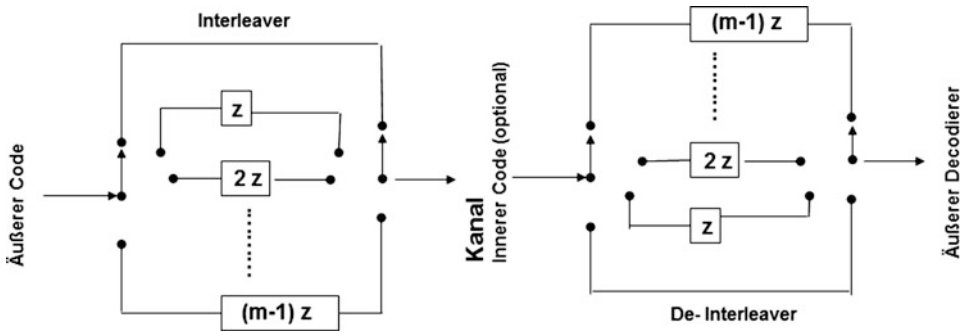


Abb. 9.24 Faltungs-Interleaver

- Der **Deinterleaver** auf der Empfangsseite ist entsprechend, aber *kopfüber* aufgebaut.
- Bei jedem Takt schalten die Multiplexer und Demultiplexer auf den nächsten Ein- bzw. Ausgang weiter. Dabei wird das nächste Zeichen (d. h. ein Bit oder ein Zeichen des dem Blockcode zugrunde liegenden Alphabets) in das gerade aktuelle Schieberegister eingelesen und das letzte Zeichen am Ausgang des Schieberegisters ausgelesen; beim Interleaver wird dieses dann auch über den Kanal gesendet. Sofern die Direktverschaltung gerade aktiv ist, wird das Zeichen durchgeschoben.
- Zu Beginn eines jeden Datenpakets der Länge n (z. B. ein Codewort des äußeren Blockcodes) müssen sämtliche Multiplexer und Demultiplexer synchronisiert sein, d. h. in der Anfangsstellung ganz oben stehen. Insbesondere muss daher die Tiefe m ein Teiler von n sein, wie oben gefordert.

Was erreicht man nun damit?

- *Benachbarte* Zeichen innerhalb eines Datenpakets mit Abstand $r < m$ werden so übertragen, dass sie jeweils Abstand größer als $rmz \geq rm(n/m) = rn$ haben. Diese Spreizung ermöglicht einem äußeren Blockcode der Länge n die Korrektur von Fehlerbündeln.
- Aber Zeichen mit Abstand m werden beim Interleaver auch *nur* mit Abstand m ausgegeben und über den Kanal übertragen.
- Durch den reziproken Aufbau des Deinterleavers werden die gespreizten Folgen wieder in Form der ursprünglichen Blöcke der Länge n zusammengesetzt, auf denen dann die Decodierung des äußeren Codes aufsetzen kann.
- Die Gesamtverzögerung eines jeden Zeichens am Ausgang beträgt $m(m-1)z$.

Schauen wir uns zum Vergleich noch mal einen Block-Interleaver an (s. Abschn. 5.4). Dort werden ja die Datenpakete der Länge n (in der Regel Codewörter eines äußeren Blockcodes) zeilenweise in die Interleaving-Matrix eingelesen. Bei Interleaving-Tiefe m werden dann die Spalten der Länge m sukzessive wieder ausgelesen und gesendet. Beim Empfang geschieht das Prozedere umgekehrt. Beim Block-Interleaver werden dabei also zwei benachbarte Symbole *nur* auf Abstand m gespreizt, nichtbenachbarte Symbole aber auf entsprechende Vielfache von m. Der Verzögerungsfaktor bei Block-Interleavern ist allerdings schlecht. Beim Empfänger kommt nämlich jedes Symbol fast $2mn$-fach verzögert an. Der Faltungs-Interleaver bietet dagegen einen etwas besseren Verzögerungsfaktor von $m(m-1)z \geq (m-1)n$. Das Problem der Verzögerung von Block-Interleavern hat man aber z. B. bei der Audio-CD durch treppenförmige Ablage und damit implizit durch Faltungs-Interleaving der Tiefe $m = n = 28$ und Basisverzögerung $z = 4$ gemildert. Faltungs-Interleaver können – wie auch Block-Interleaver – in Kombination mit oder ohne einen inneren Code genutzt werden. Bei Verwendung eines inneren und äußeren Codes hatten wir bei Block-Interleaving dann von Cross-Interleaving gesprochen. In der Praxis verwendet man bei Faltungs-Interleavern häufig einen Faltungscode als inneren Code.

9.5.4 Beispiel: Faltungs-Interleaver

Der erweiterte kleine binäre Hamming-Code $Ham_2(3)^\wedge$
Wir betrachten die Funktion des Faltungs-Interleavers noch an einem konkreten kleinen Beispiel, wir verwenden nämlich als äußeren Code den erweiterten kleinen binären Hamming-Code $Ham_2(3)^\wedge$ mit Parametern $[8, 4, 4]_2$. Als Interleaving-Tiefe wählen wir $m = 4$ und als Basisverzögerung $z = 2$; das ist möglich wegen $n = 8 = 4 \cdot 2 = mz$. Die Schieberegister im Interleaver haben folglich von oben nach unten die Länge 0, 2, 4 und 6 und im Deinterleaver umgekehrt. Wir wollen zwei Codewörter $(x_1, \ldots, x_8)$ und $(y_1, \ldots, y_8)$ in den Faltungs-Interleaver einspeisen und sehen, was nach jeweils vier Takten im Interleaver, Deinterleaver und im Output steht. Die Schieberegisterzellen seien alle mit 0 initialisiert. Nach Eingabe der beiden Codewörter schieben wir auch wieder 0 nach. Die Abb. 9.25 gibt den Verlauf der Übertragung der beiden Codewörter wieder.

Betrachten wir z. B. die Bits x_1 und $x_3$von Abstand $r = 2$, so geht x_1 mit dem 1. Takt über den Kanal, x_3 aber mit dem 19. Takt, d. h. mit einem Abstand von $18 > 2 \cdot 8 = rn$. Außerdem kommt x_1 erst mit dem 25. Takt im Output an, also mit einer Verzögerung von $24 = 4 \cdot 3 \cdot 2 = m(m-1)z$.

9.6 *Hybrid*verfahren bei Fernsehen, DSL, Mobilfunk und GPS

Codierverfahren, bei denen Codes aus den beiden unterschiedlichen Welten der Block- und Faltungscodes zum Einsatz kommen, nennt man auch ***Hybrid*verfahren**. Das *klassische* Beispiel der NASA-Codes haben wir in Abschn. 9.3 bereits ausführlich besprochen. Decodiert wird der Faltungscode auch bei der NASA mit dem Viterbi-Algorithmus. Dieser kann evtl. auftretende Fehlerbündel (Bursts) zunächst nicht alle decodieren, andererseits produziert er – wie bereits angesprochen – bei etwaigen Fehlkorrekturen meist zusätzliche Bursts. Diese werden anschließend mithilfe des Interleavers wieder auf mehrere Codewörter gespreizt, sodass der Reed-Solomon-Code die meisten der verbliebenen *vereinzelten* Fehler endgültig beseitigen kann.

9.6.1 Anwendung: Digitalfernsehen DVB der 1. Generation

In Abschn. 8.3 haben wir den Standard **DVB (Digital Video Broadcasting)** für das Digitalfernsehen vorgestellt sowie die Verkettung von BCH- und LDPC-Codes vom IRA-Typ für den DVB-Standard der 2. Generation besprochen. Wenn man bedenkt, dass Codes vom IRA-Typ eigentlich *verkappte* Faltungs-/Turbocodes sind, so handelt es sich auch hierbei um eine Art *Hybrid*verfahren.

Jetzt wollen wir aber noch auf das Fehlerkorrekturverfahren bei der 1. Generation des DVB eingehen.

Takt	Input				Interleaver						Kanal	De-Interleaver						Output			
INIT	y5	y1	x5	x1								0	0	0	0	0	0				
	y6	y2	x6	x2					0	0		0	0	0	0						
	y7	y3	x7	x3			0	0	0	0		0	0								
	y8	y4	x8	x4	0	0	0	0	0	0											
1-4	0	y5	y1	x5								x1	0	0	0	0	0				
	0	y6	y2	x6					x2	0		0	0	0	0						
	0	y7	y3	x7			x3	0	0	0		0	0								
	0	y8	y4	x8	x4	0	0	0	0	0											
5-8	0	0	y5	y1								x5	x1	0	0	0	0				
	0	0	y6	y2					x6	x2		0	0	0	0						
	0	0	y7	y3			x7	x3	0	0		0	0								
	0	0	y8	y4	x8	x4	0	0	0	0											
9-12	0	0	0	y5								y1	x5	x1	0	0	0				
	0	0	0	y6					y2	x6		x2	0	0	0						
	0	0	0	y7			y3	x7	x3	0		0	0								
	0	0	0	y8	y4	x8	x4	0	0	0											
13-16	0	0	0	0								y5	y1	x5	x1	0	0				
	0	0	0	0					y6	y2		x6	x2	0	0						
	0	0	0	0			y7	y3	x7	x3		0	0								
	0	0	0	0	y8	y4	x8	x4	0	0											
17-20	0	0	0	0								0	y5	y1	x5	x1	0				
	0	0	0	0					0	y6		y2	x6	x2	0						
	0	0	0	0			0	y7	y3	x7		x3	0								
	0	0	0	0	0	y8	y4	x8	x4	0											
21-24	0	0	0	0								0	0	y5	y1	x5	x1				
	0	0	0	0					0	0		y6	y2	x6	x2						
	0	0	0	0			0	0	y7	y3		x7	x3								
	0	0	0	0	0	0	y8	y4	x8	x4											
25-28	0	0	0	0								0	0	0	y5	y1	x5	x1			
	0	0	0	0					0	0		0	y6	y2	x6			x2			
	0	0	0	0			0	0	0	y7		y3	x7					x3			
	0	0	0	0	0	0	0	y8	y4	x8								x4			
29-32	0	0	0	0								0	0	0	0	y5	y1	x5	x1		
	0	0	0	0					0	0		0	0	y6	y2			x6	x2		
	0	0	0	0			0	0	0	0		y7	y3					x7	x3		
	0	0	0	0	0	0	0	0	y8	y4								x8	x4		
33-36	0	0	0	0								0	0	0	0	0	y5	y1	x5	x1	
	0	0	0	0					0	0		0	0	0	y6			y2	x6	x2	
	0	0	0	0			0	0	0	0		0	y7					y3	x7	x3	
	0	0	0	0	0	0	0	0	0	y8								y4	x8	x4	
37-40	0	0	0	0								0	0	0	0	0	0	y5	y1	x5	x1
	0	0	0	0					0	0		0	0	0	0			y6	y2	x6	x2
	0	0	0	0			0	0	0	0		0	0					y7	y3	x7	x3
	0	0	0	0	0	0	0	0	0	0								y8	y4	x8	x4

Abb. 9.25 Faltungs-Interleaver am Beispiel des erweiterten kleinen binären Hamming-Codes

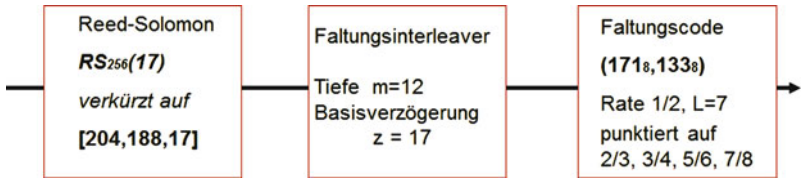


Abb. 9.26 Codierungsschema bei DVB der 1. Generation

- DVB-S: Übertragung via Satellit
- DVB-C: Übertragung via Kabel
- DVB-T: Übertragung via terrestrische Sender

DVB benutzt den Standard **MPEG2** als Quellencodierung zur Datenkompression. Da ein MPEG2-Transportpaket genau 188 Bytes umfasst, kann es als ein Wort der Länge $k = 188$ über dem Körper K mit $2^8 = 256$ Elementen gelesen werden. Daher nutzt man den Reed-Solomon-Code $RS_{256}(17)$ und verkürzt ihn auf einen MDS-Code mit Parameter $[204, 188, 17]_{256}$ (s. Abschn. 5.4). Mit diesem verkürzten Reed-Solomon-Code bildet man dann jedes MPEG2-Transportpaket von DVS auf ein Codewort der Länge $n = 204$ ab. Dieser Code wiederum wird einem Faltungs-Interleaver mit Tiefe $m = 12$ und Basisverzögerung $z = 17$ unterworfen. Das geht, weil $mz = 12 \cdot 17 = 204 = n$ ist. Der Output des Faltungs-Interleavers wird nun noch dem Faltungscode $(171_8, 133_8)$ von Rate $1/2$ und Einflusslänge $L = 7$ unterzogen. Der DVB-Standard lässt auch zu, dass angepasst an die Kanalgegebenheiten der Faltungscode auch punktiert eingesetzt werden kann, und zwar mit den Raten $2/3, 3/4, 5/6$ oder $7/8$. Die zugehörigen Punktierungsmatrizen haben wir bereits im letzten Abschnitt angegeben. Das gesamte Verfahren ist in Abb. 9.26 visualisiert. Genau genommen bezieht sich diese Beschreibung nur auf DVB-S und DVB-T. Da bei Übertragung via Kabel das Verhältnis zwischen Signal- und Rauschleistung um Größenordnungen besser ist, wird bei DVB-C auf einen Faltungscode verzichtet.

Zum Decodieren wird bei DVB-S und DVB-T zunächst auf den empfangenen Datenstrom der Viterbi-Algorithmus angewandt und dann werden per Interleaving die Fehlerbündel auf mehrere Codewörter im verkürzten Reed-Solomon-Code verteilt, sodass dieser schließlich die übrigen einzelnen Fehler korrigieren kann.

9.6.2 Anwendung: Telefonie und schnelles Internet mit DSL

DSL, Digital Subscriber Line (dt. digitaler Teilnehmeranschluss), ist ein Protokollstandard der OSI-Bitübertragungsschicht ursprünglich aus den 1990er-Jahren, bei dem Daten mit hohen Übertragungsraten über Telefonleitungen (aus Kupfer oder Glasfaser) gesendet und empfangen werden können. Die eigentliche Verbindung wird über andere Protokolle höherer OSI-Schichten hergestellt, z. B. Ethernet (Sicherungsschicht) und IP (Vermitt-

lungsschicht). Durch Nutzung eines größeren Frequenzbandes zur Datenübetragung wird eine wesentlich höhere Rate z. B. gegenüber herkömmlichen ISDN-Verbindungen erreicht und somit die wesentlichste Voraussetzung für *schnelles Internet* geschaffen. Die DSL-Variante **ADSL (Asymmetric DSL)** ist üblicherweise bei Privatkunden im Einsatz, womit DSL parallel zum normalen Telefon genutzt werden kann. Das Fehlerkorrekturverfahren von ADSL-Frames ist wie folgt aufgebaut.

- Es beruht in erster Linie auf einem Reed-Solomon-Code $RS_{256}(d)$, der vom Betreiber auf unterschiedliche Framelängen $n \leq 255$ Bytes verkürzt und mit unterschiedlichen Minimalabständen $3 \leq d \leq 25$ konfiguriert werden kann. Somit ergeben sich zwischen 2 und 24 redundante Bytes.
- Die Codewörter $(c_0, \ldots, c_{n-1})$ werden anschließend einem **Faltungs-Interleaving** der Tiefe $m = n$ und Basisverzögerung z unterzogen, wobei $z + 1$ eine 2-Potenz bis maximal 2^9 sein kann. Dabei ist auch $z = 0$ zulässig, sodass Interleaving dabei nur eine Option ist. Für alle anderen Werte von z gilt jedenfalls $z \geq 1 = n/m$.
- Eine weitere Option beim Fehlerkorrekturverfahren ist ein zusätzlicher Faltungscode als innerer Code. Dabei handelt es sich um den recht komplizierten, rekursiven Code von **Wei**, auf den wir hier aber nicht weiter eingehen wollen.

Aber auch dieses FEC-Verfahren wird zur zusätzlichen Sicherheit noch um ein ARQ-Verfahren ergänzt. Für jeden sog. Superframe, der seinerseits aus 68 Frames besteht, wird zu einigen wichtigen, klar festgelegten Bits zusätzlich eine achtstellige FCS (Frame Checking Sequence) mithilfe des Polynoms CRC-8-SAE-J1850 generiert (vgl. Abschn. 6.5), welche beim Empfang überprüft wird.

9.6.3 Anwendung: Mobilfunk mit GSM, UMTS und LTE

GSM (Global System for Mobile Communications) ist ein Standard für digitale Mobilfunknetze, der hauptsächlich für Telefonie und Kurzmitteilungen (SMS) konzipiert wurde. Es ist der Standard der sog. **zweiten Generation (2G)** als Nachfolger der analogen Systeme der ersten Generation (A-, B- und C-Netze). GSM ist die technische Grundlage der D- und E-Netze, welche in Deutschland 1992 kommerziell eingeführt wurden und zu einer raschen Verbreitung von Mobiltelefonen führten. Der GSM-Standard wird heute in rund 200 Ländern der Welt als Mobilfunkstandard genutzt, was einem Anteil von über 70 % aller Mobilfunkkunden entspricht. Es existieren später hinzugekommene Erweiterungen, wie z. B. **EDGE**, mit dem durch eine neue Modulation eine Erhöhung der Datenrate ermöglicht wird. Die GSM-Standardisierung selbst wurde von der **ETSI** (**Europäisches Institut für Telekommunikationsnormen**) in den späten 1980er-Jahren vorgenommen. Im Laufe der 1990er-Jahre wurde dann **UMTS (Universal Mobile Telecommunications System)** mit deutlich höheren Datenübertragungsraten entwickelt und ebenfalls von der ETSI in Zusammenarbeit mit dem UN-Gremium **ITU** (**International Telecommunicati-**

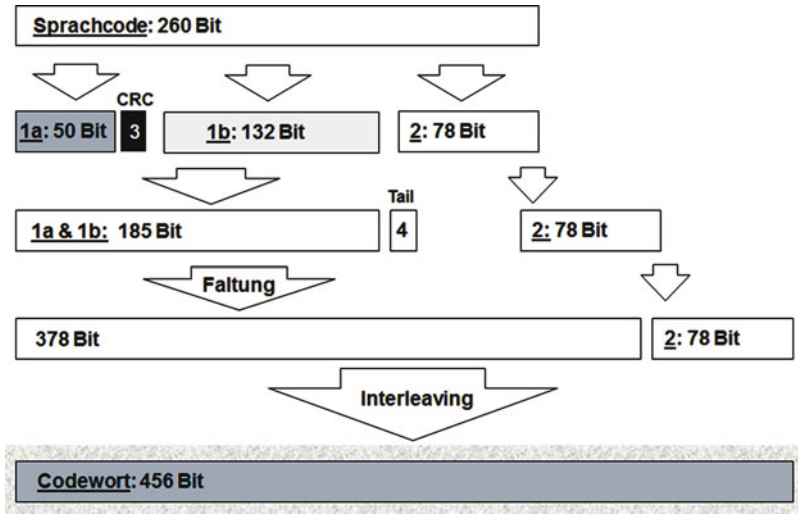


Abb. 9.27 Datenstruktur und Codierungsschema bei GSM FR (auf Basis [LNT, ETGSM])

on Union) als Mobilfunkstandard der **dritten Generation (3G)** normiert. UMTS umfasst zusätzliche Dienste, wie z. B. E-Mail, Videotelefonie, Internetzugang und Electronic Banking. Es ist seit 2004 in Deutschland kommerziell verfügbar und es gibt mittlerweile UMTS-Netze in über 100 Ländern. Die Betreuung von GSM und UMTS wurde mittlerweile an **3GPP (3rd Generation Partnership Project)** übergeben. Dies ist eine 1998 gegründete weltweite Kooperation von Standardisierungsgremien im Mobilfunkbereich, der auch die ETSI angehört. Über 3GPP ist auch ein Großteil aller Netzbetreiber, Gerätehersteller und Regulierungsbehörden als Partner organisiert. In jüngerer Zeit hat 3GPP auch bereits **LTE (Long Term Evolution)** als Mobilfunkstandard der **vierten Generation (4G)** auf den Weg gebracht. Er bietet deutlich höhere Durchsatzraten als die älteren Standards, wobei jedoch das Grundschema von UMTS im Wesentlichen beibehalten wird. So ist eine rasche und kostengünstige Nachrüstung der Infrastrukturen von UMTS- auf LTE-Technologie möglich. 2010 wurden in Deutschland die ersten LTE-Lizenzen versteigert und die ersten LTE-Sendemaste in Betrieb genommen.

Bezüglich Fehlererkennung und -korrektur beginnen wir beim GSM und beschreiben zunächst das *Hybrid*verfahren für den sog. **GSM Full Rate Code (FR)** anhand von Abb. 9.27. Dabei umfasst ein Sprachpaket 260 Bits an Nutzdaten. Die 260 Bits werden dabei in drei Klassen eingeteilt dementsprechend wie stark sich ein Bitfehler auf das Sprachsignal auswirken würde.

Die Klasse Ia umfasst 50 Bits, sie ist am stärksten zu schützen. Dies geschieht mit dem CRC-Polynom $g(x) = x^3 + x + 1$. Aus Abschn. 6.1 wissen wir, dass das Polynom $g(x)$ einen binären Code der Länge 7 erzeugt, nämlich $Ham_2(3)$. Da das Polynom $x^7 + 1$ das Polynom $x^{63} + 1$ teilt, teilt auch $g(x)$ das Polynom $x^{63} + 1$, es erzeugt daher auch einen

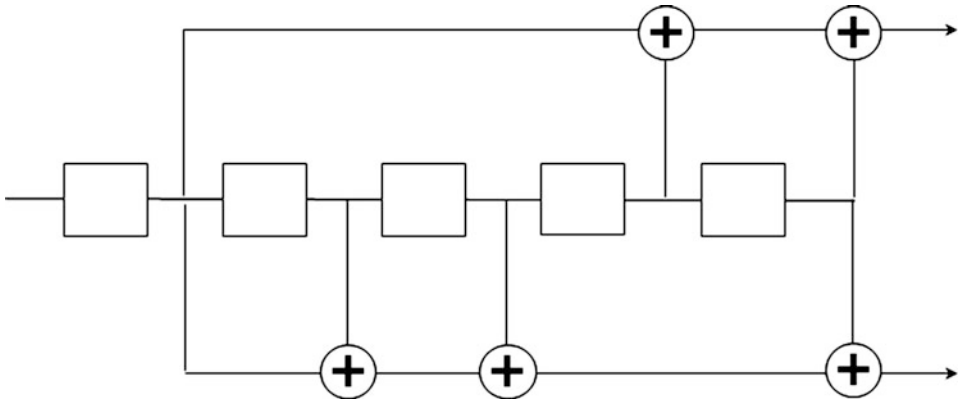


Abb. 9.28 Schieberegisterschaltbild zum Faltungscode $(23_8, 35_8)$

binären Code der Länge 63. Als CRC-Polynom kann es folglich die 50 Klasse-Ia-Bits mit einer dreistelligen FCS versehen und dabei Bündelfehler bis zur Länge 3 erkennen. Allerdings wird bei GSM nach einer CRC-Fehlererkennung kein ARQ (Automatic Repeat Request) abgesetzt, sondern es erfolgt eine automatische Fehlerverdeckung durch geeignete Interpolation. Die Klasse Ib umfasst 132 Bits, sie ist etwas weniger schützenswert. Sie wird zusammen mit den $50 + 3$ Bits der Klasse Ia einem Faltungscode mit Einflusslänge $L = 5$ und Rate $1/2$ unterworfen. Um den Faltungscode zu terminieren, werden der Sequenz von 185 Bits vier weitere Nullen als Tail-Bits angehängt, sodass der Faltungscodierer insgesamt 378 Bits ausgibt. Es handelt sich dabei um den Faltungscode $(23_8, 35_8)$, binär $10'011$ und $11'101$ und mit freiem Abstand $d_f = 7$. Er findet sich auch unter den besten Faltungscodes gemäß Tab. 9.1 (s. Abschn. 9.3). Abb. 9.28 zeigt das zugehörige Schaltbild.

Die Klasse 2 letztlich umfasst 78 Bits, diese werden ungeschützt übertragen. Somit werden aus 260 Bits Nutzdaten 456 Bits teilweise fehlergeschützte Daten. Diese 456 Bits werden aber letztlich noch einem Interleaving unterzogen. Dabei handelt es sich um ein Verfahren, das wir bislang noch nicht explizit angesprochen haben, das sog. **Diagonal-Interleaving**. Allgemeiner als beim verzögerten Block-Interleaving bzw. Faltungs-Interleaving werden beim Diagonal-Interleaving aufeinanderfolgende Codewörter noch kunstvoller ineinander verschachtelt. Konkret wird hier ein Wort mit 456 Bits über acht Interleaving-Blöcke von je 114 Bits verteilt, wobei nach jedem vierten Interleaving-Block mit einem neuen Codewort begonnen wird. Ein solcher Interleaving-Block enthält also genau 57 Bits von *einem* Codewort und 57 Bits von dem *nächsten* Codewort, wobei die Bits des Ersteren genau die ungeraden Stellen und die des Letzteren genau die geraden Stellen besetzten, aber auch zusätzlich noch trickreich gespreizt sind. Wir haben das Verfahren in Abb. 9.29 visualisiert, wollen es aber nicht genauer beschreiben.

Nach dem Deinterleaving im Empfänger werden kurze Störungen (Burst) wieder entzerrt, sodass sie entweder durch den Faltungscode korrigiert werden können oder sich

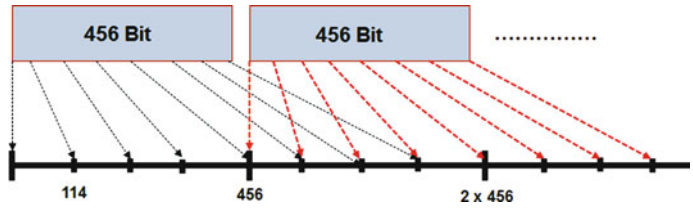


Abb. 9.29 Diagonal-Interleaving im Codierungsschema von GSM FR (auf Basis [LNT, ETGSM])

im ungeschützten Bereich nur noch gering auswirken. Durch die Kombination der unterschiedlichen Fehlerschutzverfahren im GSM, wird – obwohl der Funkkanal äußerst störanfällig ist – eine überraschend gute Sprachqualität erreicht.

Beim **GSM Enhanced Full Rate Code (EFR)** wird das für FR beschriebene *Hybrid*verfahren noch etwas erweitert, indem man eine weitere Korrekturstufe vorschaltet. Man startet dabei mit einem Sprachpaket von nur 244 Bits an Nutzdaten. Dabei werden die 50 Klasse-1a-Bits und weitere 15 Klasse-1b-Bits, also insgesamt 65 Bits, mit einer CRC-Frame Checking Sequence von 8 Bits versehen, welche vom CRC-Polynom CRC-8-SAE-J1850 erzeugt wird. Für die Definition des Polynoms vergleiche man Abschn. 6.5. Es erkennt dabei Bündelfehler bis zu einer Länge 8. Außerdem werden noch 4 ganz besonders wichtige Bits zweimal wiederholt, was also einem kleinen Wiederholungscode entspricht. Dies ergibt insgesamt $8 + 2 \cdot 4 = 16$ zusätzliche Bits, sodass man auf ein Paket von 260 Bits kommt. Auf dieses wird dann das FR-Verfahren angewandt. Damit wird eine gegenüber dem Full-Rate-Code noch deutlich bessere Sprachqualität erreicht. Nach CRC-Fehlererkennung erfolgt eine automatische Fehlerverdeckung durch geeignete Interpolation. CRC dient also dabei als Codeschutz für die Decodierung des Faltungscodes, zumindest was die wichtigsten Bits betrifft.

Wir kommen nun zur Fehlererkennung und -korrektur bei UMTS und LTE. Beide Standards beinhalten ebenfalls *Hybrid*verfahren, ähnlich denen für GSM. Sie sind jedoch flexibler angelegt, um unterschiedliche Datenraten berücksichtigen zu können. So können als äußerer Code diverse CRC-8-, CRC-12-, CRC-16- und CRC-24-Verfahren zum Einsatz kommen, die wir nicht alle in Abschn. 6.5 aufgelistet haben. Als innerer Code verwendet man die in Abb. 9.30 dargestellten Faltungscodes $(561_8, 753_8)$ und $(557_8, 663_8.711_8)$ mit Einflusslänge $L = 9$ und Rate $1/2$ bzw. $1/3$. Dabei findet man $(561_8, 753_8)$ mit freiem Abstand $d_f = 12$ wieder in Tab. 9.1 (Abschn. 9.3) unter den besten Faltungscodes von Rate $1/2$.

Wie in Abb. 9.31 beschrieben, werden die UMTS- bzw. LTE-Datenblöcke dabei zunächst um die CRC-Bitfolge und dann um 8 Tail-Bits erweitert, um anschließend den Faltungscode terminiert anwenden zu können. Ein Transportblock des UMTS-Sprachkanals DCM besteht beispielsweise aus 164 Nutzdatenbits. Wenn diese mit CRC-16 und anschließend mit dem Faltungscode der Rate $1/3$ codiert werden, ergeben sich 564 Bits, die letztlich wieder mittels Interleaving in Teilblöcke der Länge 141 gespreizt werden.

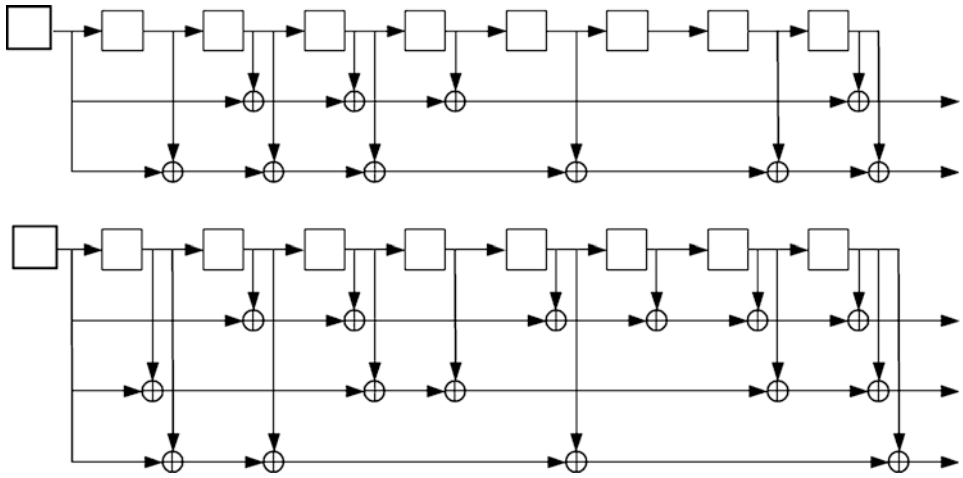


Abb. 9.30 Schieberegisterschaltbilder zu den Faltungscodes $(561_8, 753_8)$ und $(557_8, 663_8.711_8)$

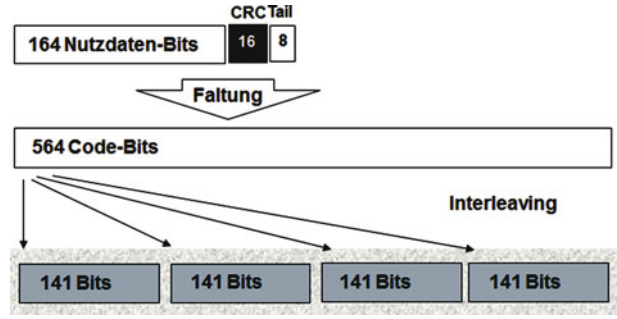


Abb. 9.31 Datenstruktur und Codierungsschema bei UMTS-DCM (auf Basis [LNT])

Allerdings haben sich bei revidierten Fassungen des UMTS- und LTE-Standards zunehmend auch Turbocodes durchgesetzt. Diese werden wir dann in Abschn. 10.3 erläutern.

9.6.4 Anwendung: Satellitennavigation mit GPS (Teil 3) und Galileo

Wir beenden hier unsere kleine Fortsetzungsreihe aus Abschn. 3.2 und 8.1 über Fehlerbehandlung bei GPS mit den Frequenzbändern L2C und L5. Während L2C ein zusätzlich zivil nutzbares Frequenzband ist (neben dem klassischen L1 C/A und dessen Nachfolger L1C), ist L5 als *Safety-of-Life*-Signal insbesondere für die Luftfahrt reserviert. Wir erläutern das Vorgehen wieder anhand von Abb. 9.32. In beiden Fällen besteht ein Datenpaket aus 20 Präambel- und ID-Bits sowie 256 weiteren Bits (u. a. 6 Tail-Bits), an das eine 24-Bits-FCS mit dem CRC-Polynom CRC-24-Q als äußerer Code angehängt wird. Wir haben im Abschn. 6.5 gesehen, dass CRC-24-Q Fehler an bis zu 3 Bits erkennen kann

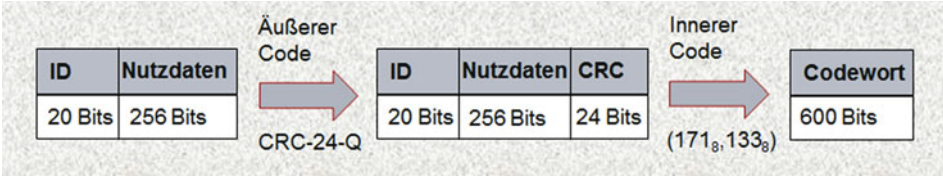


Abb. 9.32 Codierungsschema bei GPS L2C und L5 (auf Basis [Gar])

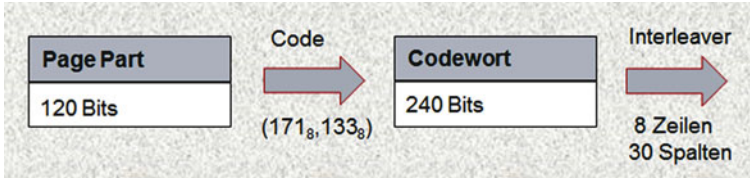


Abb. 9.33 Codierverfahren beim Satellitennavigationssystem Galileo E1-OS (auf Basis [Gar])

und außerdem jede ungerade Anzahl von Fehlern und Bursts bis zur Länge 24. Als innerer Code wird wieder der Faltungscode $(171_8, 133_8)$ terminiert verwendet, mit Rate $1/2$ und Einflusslänge $L = 7$, das Ganze allerdings ohne Interleaving.

Es handelt sich damit wieder um eine klassische *Hybrid*konfiguration. Mit dem Faltungscode werden dabei die Fehler korrigiert und das Ergebnis mit CRC-24-Q auf Richtigkeit überprüft. Gegebenenfalls muss anschließend interpoliert werden oder es wird einfach ignoriert. CRC dient also auch hier wieder als Codeschutz für die Decodierung des Faltungscodes.

Wir wollen auch noch kurz auf die europäische Satellitennavigation **Galileo** (E1-Band Open Service) eingehen. Zunächst werden hierbei sog. Pages von jeweils 240 Bits mit CRC-24-Q als äußerem Code geschützt. Wie in Abb. 9.33 zu sehen, werden anschließend jeweils zwei Page Parts mit 120 Bits ebenfalls mit dem Faltungscode $(171_8, 133_8)$ terminiert codiert, dieser String aber vor dem Senden zur Spreizung von Bursts noch einem Interleaver mit acht Zeilen und 30 Spalten unterzogen.

Der Faltungscode wird dabei sowohl bei GPS als auch bei Galileo mittels Viterbi-Algorithmus decodiert.

Turbocodes

10

Turbocodes – die neue Welt? Das Konzept der **Turbocodes** wurde erstmalig 1993 von **Claude Berrou**, **Alain Glavieux** und **Punya Thitimajshima** vorgeschlagen, aber zunächst nicht so recht ernst genommen, da man ihre Güte auch nur mit Simulationen ermitteln und sie durch Pseudozufallsverfahren optimieren konnte. Wir erfahren, dass man heute unter einem Turbocode präziser zwei gleiche, über einen Interleaver parallel geschaltete, rekursive Faltungscodierer in systematischer Form (**RSC**) versteht. Als ein konkretes Beispiel überlegen wir uns das Schaltbild unseres **RA-Blockcodes** und bringen dies in die Form eines Turbocodes.

SOVA- und BCJR-Algorithmus Turbocodes zählen heute zu den besten fehlerkorrigierenden Codes, was vor allem aber ihrer iterativen Decodiermethode geschuldet ist. Wir lernen, dass hierbei sog. **Soft-Decision**-Verfahren zum Einsatz kommen, bei denen das Decodierergebnis jeweils mit *Vertrauensfaktoren* versehen ist. Hierbei handelt es sich einerseits um den **SOVA (Soft-Output-Viterbi-Algorithmus)**, vor allem aber um den **BCJR/MAP-Algorithmus (Maximum A-Posteriori Probability)**, der von Hause aus ein Soft-Decision-Verfahren ist und 1974 von **Bahl**, **Cocke**, **Jelinek** und **Raviv** vorgeschlagen wurde.

Turbocodes bei Mobilfunk und in der Raumfahrt Turbocodes haben aufgrund ihrer Güte mittlerweile auch Eingang in diverse Standards gefunden. Wir erfahren, welcher Turbocode für die **Mobilfunkstandards UMTS** (3. Generation) und **LTE** (4. Generation) vorgegeben ist. Turbocodes verschiedener Raten sind auch Teil des Raumfahrtstandards **CCSDS (Consultative Committee for Space Data Systems)**. Eingesetzt wurden und werden Turbocodes bei der ESA-Forschungsmission **SMART-1** zum Mond, bei der ESA-Sonde **Rosetta**, die ihren Lander **Philae** auf dem Kometen ***Tschuri*** abgesetzt hat, sowie bei der NASA-Sonde **New Horizons** auf ihrem Weg vorbei an **Pluto** hinaus in den Kuiper-Gürtel.

O. Manz, *Fehlerkorrigierende Codes*, DOI 10.1007/978-3-658-14652-8_10

10.1 Turbocodes – die neue Welt?

10.1.1 Was sind eigentlich Turbocodes?

Turbocodes wurden erstmalig 1993 von **Claude Berrou** (geb. 1951, französischer Elektrotechniker), **Alain Glavieux** (1949–2004, französischer Informationstheoretiker) und **Punya Thitimajshima** (1955–2006, thailändischer Informationstheoretiker) vorgeschlagen und publiziert. Auch wenn manche Autoren ursprünglich (auch in der Originalarbeit) unter Turbocodes beliebigere parallel und seriell verkettete Block- und Faltungscodes verstanden haben, so hat sich doch heute die in Abb. 10.1 wiedergegebene Konstellation herauskristallisiert, für die man den Begriff **Turbocodes** (im engeren Sinn) verwendet.

- Es handelt sich bei einem **Turbocode** um zwei **gleiche, parallel geschaltete, rekursive Faltungscodierer**, die zu jedem Inputbit x_i jeweils ein Outputbit y_i und z_i liefern.
- Die Codierung erfolgt **systematisch**, d. h., das Inputbit x_i wird pro Codewort jeweils mitausgegeben.
- Ein **Block-Interleaver** ist integraler Bestandteil des Codierers und wichtiger Hebel für die Güte des Codes. Dabei wird nach dem zeilenweisen Einlesen der Matrix eine **Permutation** der Bits vorgenommen, bevor spaltenweise ausgelesen und die Sequenz dem zweiten Faltungscodierer zugeführt wird. Dabei sind auch einzeilige Matrizen zulässig.

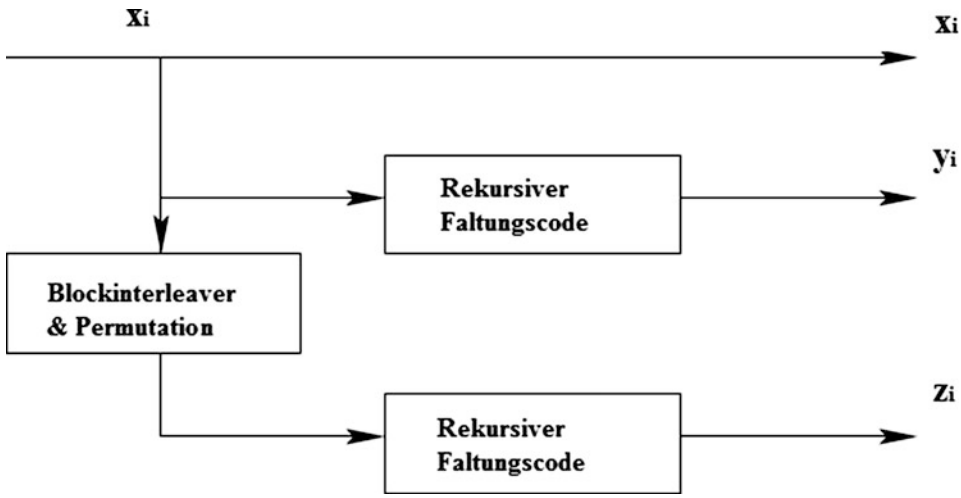


Abb. 10.1 Grundschema eines Turbocodes

Die einzelnen Outputs werden wie bei gewöhnlichen Faltungscodes mit einem Multiplexer zu Codewörtern mit jeweils 3 Bits ineinandergeschoben. Der Output der x_i sowie der Input zum ersten Faltungscodierer werden so lange gepuffert, bis der Input zum zweiten Faltungscodierer phasengleich erfolgt. Ziel der Permutation ist, dass dort, wo ein Teilstring der y_i kleines Gewicht hat, der entsprechende String der z_i großes Gewicht haben sollte. Diese Konstellation erzeugt allerdings immer nur einen Code der Rate $1/3$. Um auch andere Raten generieren zu können, lässt man darüber hinaus zu, dass die beiden rekursiven Faltungscodierer mehr als nur einen Output y_i und z_i liefern. Wir hatten in Abschn. 9.1 schon **RSCs (Recursive Systematic Convolutional Codes)** angesprochen. Sie lassen sich wie auch Turbocodes terminieren. Allerdings geht dies nicht so einfach wie bei nichtrekursiven Faltungscodes mit einigen Nullen als Tail-Bits. Im Falle von rekursiven Faltungscodes hängen nämlich die Tail-Bits von den vorherigen Informationsbits und der Rückkopplung ab.

10.1.2 Performance von Turbocodes

Nun muss man allerdings auch hier eines klar festhalten: Während man für algebraische Blockcodes eine ganze Menge an mathematischen Methoden zur Verfügung hat und bereits bei nichtrekursiven Faltungscodes davon nicht mehr allzu viel übrig blieb, ist man bei Turbocodes etwa bei null angekommen. Das einzige Konstruktionsprinzip ist *Probieren und Simulieren*. Daher waren nach der Veröffentlichung viele Forschergruppen eher skeptisch, sodass der Nutzen von Turbocodes erst Jahre später anerkannt wurde. Tatsache ist nämlich, dass Turbocodes der sog. **Shannon-Grenze** sehr nahe kommen, viel näher jedenfalls als z. B. Reed-Solomon-Codes und nichtrekursive Faltungscodes. Wir wollen erklären, was hiermit gemeint ist.

Während für Blockcodes deren **Minimalabstand** wichtigstes Gütekriterium ist und bei Faltungscodes immerhin noch der **freie Abstand** als Gütekriterium dienen kann, ist bei Turbocodes davon nichts mehr übrig – nur noch das sog. **Performancediagramm**.

Wir haben dies in Abschn. 8.3 schon kurz im Zusammenhang mit LDPC-Codes von IRA-Typ erwähnt. Hier also noch mal genauer, was es mit dem Performancediagramm auf sich hat. Auf der x-Achse könnte man die **Signalleistung** auftragen, d. h. die Leistung des Trägersignals der Information. Intuitiv ist klar: Je höher diese Leistung, desto *verständlicher* sollte eine Information nach Übertragung über den Kanal beim Empfänger ankommen. Das, was im Kanal normalerweise stört, ist seine permanente **Rauschleistung**. Also wählt man auf der x-Achse das Verhältnis E_b/N_0 von **Signalleistung N_b** zu **Rauschleistung**

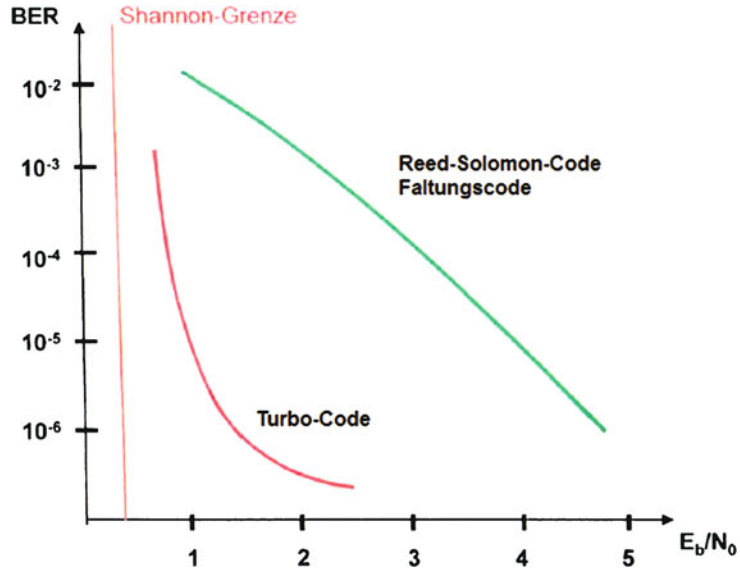


Abb. 10.2 Performancediagramm mit typischen Verläufen (auf Basis [Kla])

N_0 (in dB) und sagt: Je größer dieser Wert ist, umso weiter also die Signalleistung aus der Rauschleistung herausragt, desto *überflüssiger* sind eigentlich Fehlerkorrekturverfahren.

Auf der y-Achse trägt man **BER (Bit-Error-Ratio)** auf, das ist die gemessene **Bitfehlerrate** beim Empfänger, also die Anzahl der falschen Bits bezogen auf alle gesendeten Bits. Diese kann man z. B. auftragen für den Fall, dass kein fehlerkorrigierender Code genutzt wird, was man manchmal auch tut, um einen Vergleich zu haben. Viel interessanter ist es aber, die Kurve für codierte Information zusammen mit dem jeweiligen Decodierverfahren aufzunehmen. Es ist klar, dass sich die BER-Kurve für den gleichen Code, aber unter Verwendung unterschiedlicher Decodierverfahren unterscheiden kann.

Mit solchen Diagrammen arbeitet man also bei Turbocodes. Die Abb. 10.2 zeigt ein Diagramm mit typischen Verläufen für Turbocodes einerseits und Reed-Solomon- bzw. Faltungscodes andererseits. Man sieht daran, dass ein guter Turbocode eine deutlich bessere BER erreichen kann als z. B. Reed-Solomon-Codes und das bei ungünstigerem Signal-Rausch-Verhältnis E_b/N_0. Dies ist vergleichbar mit der Güte der IRA-Codes gemäß Abb. 8.8.

Man sieht aber noch etwas anderes an dem Diagramm. Erinnern wir uns dazu noch mal an den Satz von Shannon aus Abschn. 1.4, der besagt, dass man bei einer Coderate unterhalb der *Durchsatzkapazität* eines Kanals eine beliebig geringe Bitfehlerwahrscheinlichkeit erreichen kann. In der Sprache des Performancediagramms bedeutet das, dass man oberhalb eines bestimmten Signal-Rausch-Verhältnisses, der sog. **Shannon-Grenze** des Kanals, Codes mit beliebig kleiner BER finden kann – zumindest theoretisch. Dabei kommen Turbocodes mit ihren sehr kleinen Bitfehlerraten schon sehr nahe an diese theoretisch

mögliche Shannon-Grenze heran. Dies ist also das eigentlich attraktive und *revolutionäre* an Turbocodes.

Es sticht bei Performancediagrammen von Turbocodes noch eine andere Auffälligkeit ins Auge. Die Kurve flacht nämlich ab einer BER von ca. 10^{-6} deutlich ab. Man nennt das den **Error Floor**. Will man unter diesen BER-Bereich kommen, so muss man dem Turbocode noch einen Blockcode nachschalten – also wieder *hybrid* arbeiten.

10.1.3 Beispiel: Turbocodes

RA-Codes als Turbocodes

Wie angekündigt, kommen wir jetzt nochmals auf unsere RA-Codes zurück. Wir hatten in Abschn. 9.1 RA-Codes schon mit RSC-Codes in Verbindung gebracht, dabei fehlte aber noch das *Wiederholen* (Wdh) und *Permutieren* (Perm). Das gesamte Schaltbild eines RA-Codes sieht also wie in Abb. 10.3 dargestellt aus. Man kann sich nun überlegen (s. [HBH]), dass man diese Schaltung unter *Beibehaltung der Performance* in eine Turbocodeschaltung gemäß Abb. 10.4 überführen kann. Dabei umfasst die Box sowohl das Interleaving mit Permutation für den unteren Codierer als auch das Puffern der Eingangsbits für den oberen Codierer.

Diese Schaltung hat auch den Vorteil, dass man den Code mittels Algorithmen für Turbocodes decodieren kann. Diese sind ein wesentlicher Grund für die Güte von Turbocodes, wie wir im nächsten Abschnitt sehen werden.

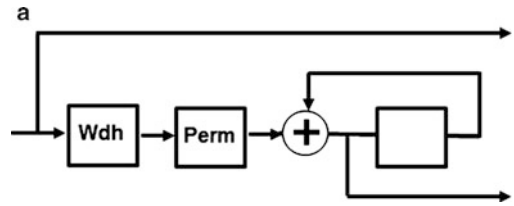


Abb. 10.3 Schaltbild eines RA-Codes

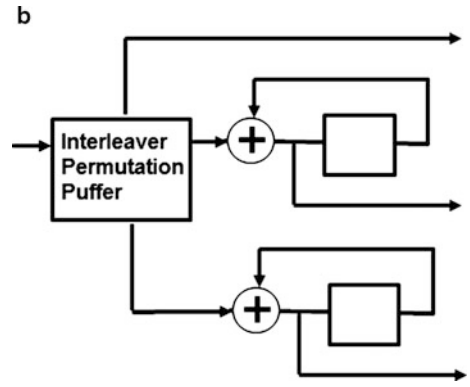


Abb. 10.4 RA-Code als Turbocode

10.2 SOVA- und BCJR-Decodierung

10.2.1 Turbodecodierer mit SOVA-Algorithmus

Wir haben im letzten Abschnitt schon gelernt, dass das Performancediagramm nicht nur vom Code selbst abhängt, sondern auch wesentlich vom Decodierverfahren geprägt wird. Würde man z. B. versuchen, einen Turbocode mit dem gewöhnlichen Viterbi-Algorithmus zu decodieren, so ergibt sich bei Weitem keine so gute Performance. Genau genommen ist also die Decodiermethode, die wir jetzt skizzieren werden, das eigentlich Besondere an einem Turbocode. Ein Turbodecodierer besteht zunächst einmal aus zwei einzelnen Decodierern. Jeder der beiden führt einen Viterbi-Algorithmus aus, allerdings nicht den gewöhnlichen, den wir schon kennen, sondern einen, der mit Wahrscheinlichkeiten operiert. Dieser sog. **SOVA-Algorithmus (Soft-Output-Viterbi-Algorithmus)** wurde erst viel später im Jahr 1989 von **J. Hagenauer** vorgeschlagen. Er benutzt zwar auch ein Trellis-Diagramm, um einen Survivor-Pfad zu bestimmen, aber im Gegensatz zum gewöhnlichen Viterbi-Algorithmus vergleicht er bei jedem Takt die Teilstücke des überlebenden Pfads mit denen der nichtüberlebenden. Wo sich überlebender Pfad und nichtüberlebende nur *wenig* unterscheiden, wird die Wahrscheinlichkeit der zugehörigen Inputbits angehoben, sonst wird sie reduziert. Dies kann und muss man anhand eines Abstandsbegriffs (**Metrik**) auch exakt fassen, was wir hier allerdings nicht tun wollen. Am Ende wird also dabei ein Survivor-Pfad bestimmt, bei dem die zugehörigen Inputbits nicht *hart* bestimmt sind, sondern jedes mit einem *Vertrauensfaktor* versehen ist, in den noch andere Gegebenheiten des Kanals eingehen können. Solche Vertrauensfaktoren kann man sich bei der **SOVA-Decodierung** etwa so vorstellen.

$$
\begin{aligned}
0_1 &= \text{sehr wahrscheinlich } 0\\
0_2 &= \text{wahrscheinlich } 0\\
0_3 &= \text{eher } 0\\
1_3 &= \text{eher } 1\\
1_2 &= \text{wahrscheinlich } 1\\
1_1 &= \text{sehr wahrscheinlich } 1
\end{aligned}
$$

Die Granularität der Vertrauensfaktoren kann dabei frei gewählt werden. Bei Granularität 1 ergibt sich der gewöhnliche Viterbi-Algorithmus. Der gesamte Turbodecodierer funktioniert wie in Abb. 10.5 und 10.6 dargestellt. Die beiden Einzeldecodierer sind mit einem Interleaver, der dem Codierinterleaver entspricht, und einem Deinterleaver, der das Interleaving rückgängig macht, miteinander verbunden. Betrachten wir zunächst den Empfang der vom Kanal möglicherweise veränderten Bitfolgen x_i', y_i' und z_i'. Decodierer 1 wird dabei mit den Sequenzen x_i' (d. h. den modifizierten Informationsbits) und y_i' versorgt. In Decodierer 2 eingelesen werden die z_i' und die mittels Interleaver permutierten x_i', um sie mit der Reihenfolge der z_i' kompatibel zu machen.

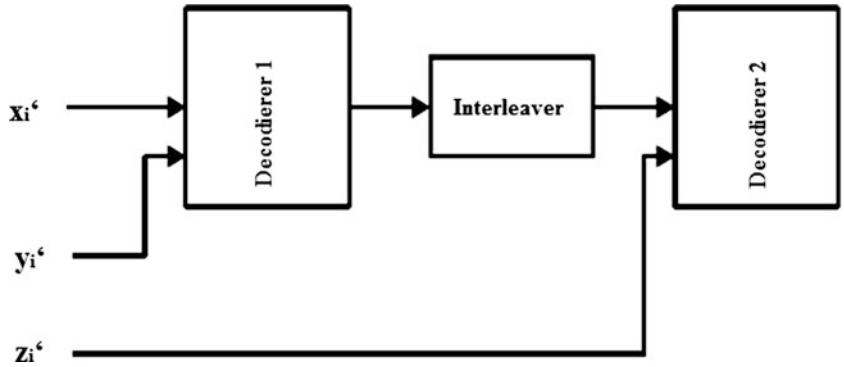


Abb. 10.5 Prinzipieller Aufbau eines Turbodecodierers in der Startphase

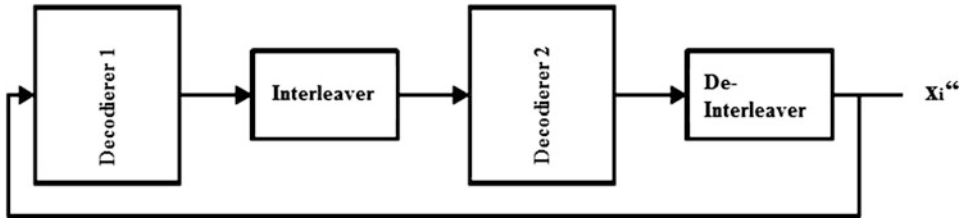


Abb. 10.6 Prinzipieller Aufbau eines Turbodecodierers in der Betriebs- und Ausgabephase

Jetzt startet zunächst Decodierer 1 seinen SOVA-Lauf und ermittelt dabei eine Inputbitsequenz mit Vertrauensfaktoren. Diese übergibt er an Decodierer 2, wobei zur Konsistenz der Reihenfolge bei der Übertragung der Interleaver durchlaufen werden muss. Nun führt Decodierer 2 seinen SOVA-Lauf durch, berücksichtigt aber dabei bereits die von Decodierer 1 ermittelten Faktoren als sog. **A-priori-Bewertung**. Das Ergebnis stellt er diesmal über den Deinterleaver wieder Decodierer 1 zur Verfügung. So geht dies einige Male hin und her, bis zu einer aus der Erfahrung heraus definierten Abbruchmarke. Am Ende steht also eine Sequenz mit Vertrauensfaktoren für jedes Bit, woraus dann als harte Entscheidung die decodierte Bitfolge x_i'' ermittelt wird.

Dieses Verfahren konvergiert in der Praxis überraschend schnell (nach 5–20 Iterationen). Dennoch ist der Rechenaufwand insgesamt nicht unerheblich, sodass dabei Antwortzeiten von bis zu zehn Sekunden herauskommen können. Für Anwendungen in der Raumfahrt spielt das keine Rolle, für Telefonverbindungen muss man solche Antwortzeiten evtl. unter Billigung geringerer Sprachqualität reduzieren. Es wird aber auch klar, dass man sich schon deshalb auf zwei gleiche, parallel geschaltete Codierer beschränkt, um die Komplexität des Decodierverfahrens nicht noch weiter ansteigen zu lassen.

10.2.2 BCJR-Algorithmus

Neben dem Viterbi-Algorithmus gibt es noch ein weiteres Verfahren, welches zum Decodieren von Faltungscodes eingesetzt wird, nämlich den **BCJR-Algorithmus**. Die Abkürzung steht dabei für **L. Bahl**, **J. Cocke**, **F. Jelinek** und **J. Raviv**, die das Verfahren im Jahr 1974 konzipiert haben. Wie wir aus Abschn. 9.4 wissen, ist der Viterbi-Algorithmus eine Maximum-Likelihood-Decodierung, d. h., er bestimmt zu einer empfangenen Bitfolge die **wahrscheinlichste Codebitsequenz**, ohne dass er dabei *Vorwissen* über die Wahrscheinlichkeit dieser Sequenz heranzieht. Genau solches Vorwissen nutzt aber der BCJR-Algorithmus. Er bestimmt nämlich aus **A-priori-Wahrscheinlichkeiten** für die Informationsbits zusammen mit der empfangenen Bitfolge als sog. **Stichprobe** *die* Inputbitsequenz mit **maximaler A-posteriori-Wahrscheinlichkeit**. Man spricht daher auch häufig von **MAP (Maximum A-Posteriori Probability)**. Man kann zeigen, dass die **BCJR-Decodierung** im Sinne der **minimalen Bitfehlerwahrscheinlichkeit** der optimale Decodieralgorithmus ist.

Der BCJR-Algorithmus kann ebenso wie der Viterbi-Algorithmus in Form eines Trellis-Diagramms grafisch dargestellt werden. Und so geht man prinzipiell bei einem Faltungscode der Rate $1/n$ vor. Sei $u_1, \ldots, u_N$ eine Inputfolge von Informationsbits und $y = y_1, \ldots, y_N$ die empfangene Folge, wobei jedes y_j aus n Bits besteht. Sei weiterhin s der Zustand im Trellis-Diagramm zum Takt j und s' der zum Takt $j-1$. Der BCJR-Algorithmus bestimmt nun für jede Position j der Inputfolge das Verhältnis der A-posteriori-Wahrscheinlichkeiten (genau genommen deren *Logarithmus*)

$$P(u_j = 1|y)/P(u_j = 0|y) = \sum_{R_1} P(s', s, y)/\sum_{R_0} P(s', s, y),$$

wobei R_1 alle Übergänge von s' nach s mit $u_j = 1$ durchläuft und R_0 alle die mit $u_j = 0$. Ist der Quotient deutlich größer als 1, so ist die Wahrscheinlichkeit für $u_j = 1$ sehr hoch, im Falle von deutlich kleiner 1 die für $u_j = 0$. Wir setzen zur Abkürzung $y = y_{<j}, y_j, y_{>j}$.

Die gemeinsame Wahrscheinlichkeit $P(s', s, y)$ für s', s und y kann man nun schreiben als

$$P(s', s, y) = \alpha_{j-1}(s')\gamma_j(s', s)\beta_j(s),$$

wobei die α, γ und β die Wahrscheinlichkeiten für die **Vergangenheit („backward“)**, **Gegenwart („present“)** und **Zukunft („forward“)** der Sequenz y bezeichnen. Genauer ist

- $\alpha_{j-1}(s') = P(s', y_{<j})$ die gemeinsame Verteilung des Vorgängerzustands s' und der Vorgängerwerte $y_{<j}$,
- $\gamma_j(s', s) = P(y_j, s|s')$ die gemeinsame Verteilung des aktuellen Werts y_j und des aktuellen Zustands s, gegeben der Vorgängerzustand s' und

- $\beta_j(s) = P(y_{>j}|s)$ die Verteilung der Nachfolgewerte $y_{>j}$, gegeben der aktuelle Zustand s.

Beim BCJR-Algorithmus geht es also stets darum, die α-, β- und γ-Wahrscheinlichkeiten für konkrete Situationen (d. h. Kanalgegebenheiten, Codierungsvorschrift, A-priori-Wahrscheinlichkeiten) zu bestimmen. Meist wird dabei die Rauschleistung N_0 des Kanals als additives weißes Gauß-verteiltes Rauschen modelliert (**AWGN, Additive White Gaussian Noise**). Dass diese Rechnungen sehr komplex sind, kann man jetzt bereits erkennen, wir verzichten daher auf konkrete Beispiele. Man sieht aber bereits hier, wie der BCJR-Algorithmus die Verbundwahrscheinlichkeiten von aufeinanderfolgenden Codebits konsequent ausnutzt.

Warum erwähnen wir den BCJR-Algorithmus an dieser Stelle? Nun, er ist aufgrund seiner Struktur ein **Soft-Decision-Verfahren** und daher prädestiniert für iterative Decodierung wie bei Turbocodes. Der Viterbi-Algorithmus hingegen wurde durch SOVA erst nachträglich zum Soft-Decision-Verfahren gemacht. Es ist daher nicht ganz verwunderlich, dass bei Turbocodes in Wirklichkeit meist mittels BCJR-Algorithmus decodiert wird und weniger mit SOVA. Bei nichtrekursiven Faltungscodes spielt dagegen BCJR eine eher untergeordnete Rolle.

10.3 Turbocodes bei Mobilfunk und in der Raumfahrt

10.3.1 Anwendung: UMTS und LTE

In Abschn. 9.6 hatten wir bereits die verschiedenen Generationen der Mobilfunktechnik vorgestellt. Wir haben auch schon gesehen, dass bei GSM (2G), UMTS (3G) und LTE (4G) standardmäßig *Hybrid*verfahren zur Fehlererkennung und -korrektur vorgesehen sind. Allerdings wird bei UMTS und LTE auch alternativ die Möglichkeit gegeben, mit einem Turbocode zu arbeiten, wie er in Abb. 10.7 dargestellt ist.

Es handelt sich dabei also um zwei gleiche, parallel geschaltete, rekursive Faltungscodes, und zwar um genau denjenigen, den wir schon als Beispiel eines RSC-Codes in Abschn. 9.1 kennengelernt haben. Dabei gibt jeweils der untere Ast die Rekursionsbedingung, d. h. die Rückkopplung, an, der obere Ast berechnet wie bei gewöhnlichen Faltungscodes das Ausgabebit. Der Code ist in systematischer Form, gibt also die Informationsbits mit aus und hat somit Rate $1/3$. Wiederum umfasst dabei die Box sowohl das Interleaving mit Permutation für den unteren Codierer als auch das Puffern der Eingangsbits für den oberen Codierer.

Der Turbocode kann nach jeweils t Takten terminiert werden, wobei die zulässigen t im Standard zwischen 40 und 5114 festgelegt sind. An das gewählte t ist dann auch die für das Interleaving zu nutzende Matrix angepasst. Bei $t = 40$ handelt es sich beispielsweise um eine Matrix $M = (m_{ij})$ mit fünf Zeilen und acht Spalten. Die Bits werden zunächst zeilenweise eingelesen und anschließend innerhalb der Matrix wie folgt permutiert, bevor

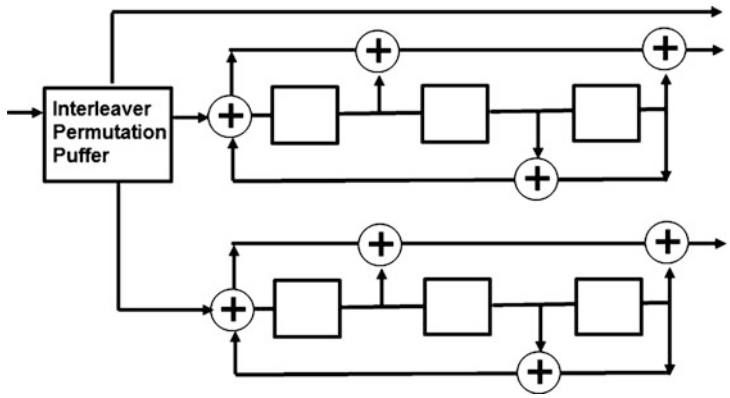


Abb. 10.7 Turbocode für UMTS und LTE

sie wieder spaltenweise ausgelesen werden:

$$\begin{pmatrix} m_{11} & \dots & m_{18} \\ \vdots & & \vdots \\ \vdots & & \vdots \\ \vdots & & \vdots \\ \vdots & & \vdots \\ m_{51} & \dots & m_{58} \end{pmatrix} \Longrightarrow \begin{pmatrix} m_{58} & m_{54} & m_{53} & m_{57} & m_{55} & m_{56} & m_{51} & m_{52} \\ m_{42} & m_{44} & m_{43} & m_{47} & m_{45} & m_{46} & m_{41} & m_{48} \\ m_{32} & m_{36} & m_{35} & m_{37} & m_{32} & m_{34} & m_{31} & m_{38} \\ m_{22} & m_{24} & m_{23} & m_{27} & m_{25} & m_{26} & m_{21} & m_{28} \\ m_{12} & m_{16} & m_{15} & m_{17} & m_{12} & m_{14} & m_{11} & m_{18} \end{pmatrix}.$$

Zur Illustration ist in Abb. 10.8 auch noch das Performancediagramm für verschiedene Terminierungen t sowie bei zehn Iterationsläufen des Decodierers angegeben. Je größer t, umso besser der Code, aber auch umso länger die Laufzeit.

10.3.2 Die CCSDS-Standards

Das **CCSDS (Consultative Committee for Space Data Systems)** mit Sitz in Washington, D.C., ist eine internationale Organisation, in der sich die führenden Weltraumorganisationen zusammengefunden haben. Die Aufgabe des CCSDS ist die Ausarbeitung gemeinsamer Methoden des Datenverkehrs mit Raumsonden. CCSDS wurde im Jahr 1982 auf Anregung einer gemeinsamen Arbeitsgruppe von **NASA** und der **Europäischen Weltraumorganisation ESA** gegründet. Weitere Mitglieder sind u. a. die FSA (Federal Space Agency, Russland) sowie die CAST (Chinese Agency for Space Technology) mit Beobachterstatus. Eine immer weiter ausgebaute Zusammenarbeit zwischen den einzelnen Raumfahrtorganisationen und die erforderliche gegenseitige Nutzung von Infrastrukturen machte die Festlegung gemeinsamer Standards zwingend erforderlich. Zu den Standards gehören z. B. einheitliche Zeit- und Positionsangaben sowie natürlich auch die Normung

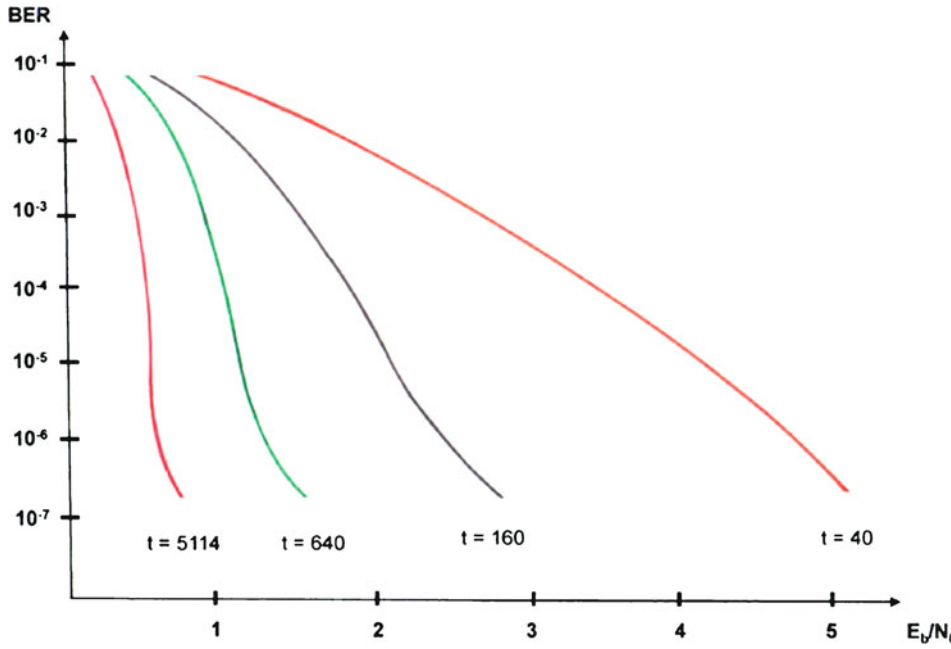


Abb. 10.8 Performancediagramm für verschiedene Parameter t des Turbocodes für UMTS/LTE (auf Basis [Joo])

von Codierverfahren. Der Standard enthält Aussagen über Blockcodes, Faltungscodes, LDPC-Codes und Turbocodes.

So besagt der CCSDS-Standard, dass beim NASA Concatenated Planetary Standard Code (s. Abschn. 9.3) auch folgende Variationen möglich wären:

- Statt des 16-fehlerkorrigierenden Reed-Solomon-Codes $RS_{256}(33)$ mit Parameter $[255, 223, 33]$ ist auch der 8-fehlerkorrigierende $RS_{256}(17)$ mit Parameter $[255, 239, 17]$ zulässig. Ergänzend hierzu noch eine Anmerkung. Wie wir ja wissen, wird der jeweils zugrunde liegende Körper K mit $2^8 = 256$ Elementen von einem irreduziblen Polynom vom Grad 8 erzeugt. Davon gibt es allerdings mehrere und alle sind im Prinzip *gleich gut*, wenngleich sich aber die zugehörigen Multiplikationstabellen formal unterscheiden. Daher gehört genau genommen zu jedem Reed-Solomon-Code auch die Angabe, welches Polynom dem Körper zugrunde liegt. Wir haben dies nicht immer explizit getan, deshalb sei hier einmal der Vollständigkeit halber erwähnt, dass der CCSDS-Standard als irreduzibles Polynom $f(x) = x^8 + x^7 + x^2 + x + 1$ vorgibt.
- Die Tiefe des Block-Interleavers kann $1, 2, 3, 4, 5$ oder 8 sein, wegen Tiefe 1 ist also auch kein Interleaving grundsätzlich zulässig.

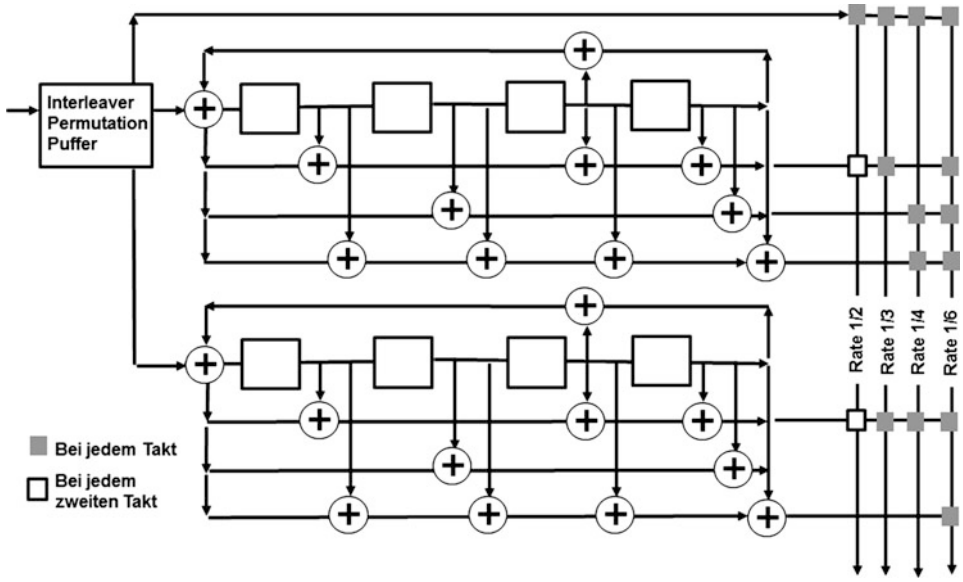


Abb. 10.9 Turbocode des CCSDS-Standards (auf Basis [CCS1])

- Der Standard Convolutional Code $(171_8, 133_8)$ kann auch angepasst an die jeweiligen Kanalbedingungen auf die Raten $2/3, 3/4, 5/6$ und $7/8$ punktiert werden. Die zugehörigen Punktierungsmatrizen haben wir bereits in Abschn. 9.5 aufgelistet.

Im Übrigen ist dies dann auch der Standard der ESA.

Was Turbocodes betrifft, so nennt der CCSDS-Standard die in Abb. 10.9 dargestellte Konfiguration, wobei hier jeweils die obere Verschaltung rückgekoppelt ist. Man sieht daran, dass es die Optionen gibt, Turbocodes der Raten $1/2, 1/3, 1/4$ oder $1/6$ zu wählen, je nachdem welche Ausgänge man schaltet. Auf das dezidierte Interleaving- und Permutationsschema für den unteren Codierer, das der CCSDS-Standard vorgibt, wollen wir hier aber nicht weiter eingehen.

Die Anzahl der Informationsbits bis zur nächsten Terminierung ist im CCSDS-Standard auf $1784, 3568, 7136$ oder 8920 festgelegt, also jeweils Vielfache von $8 \cdot 223 = 1784$. Damit ist die Blocklänge grundsätzlich kompatibel zu einer alternativen Reed-Solomon-Codierung mit $RS_{256}(33)$.

Interessanterweise empfiehlt der Standard aber auch, dem Turbocode zur Fehlererkennung ein geeignetes CRC-Verfahren vorzuschalten (Zitat [CCS1], S. 6-1): „When Turbo codes are used ..., the Frame Error Control Field (FECF) ... shall be used to validate the Transfer Frame. NOTE – While providing outstanding coding gain, Turbo codes may still leave some undetected errors in the decoded output." Für die Berechnung des FECF-Wertes empfielt CCSDS in [CCS2] ein CRC-Verfahren mit dem Polynom CRC-16-CCITT.

10.3.3 Anwendung: ESA-Raumfahrt – SMART-1 und Rosetta

Die **Europäische Weltraumorganisation ESA** (**European Space Agency**) ist eine 1975 gegründete, internationale Weltraumorganisation mit Sitz in Paris. Ziel ist eine nachhaltige Stärkung der europäischen Raumfahrtaktivitäten, insbesondere um den technologischen Rückstand gegenüber Raumfahrtnationen wie Russland und USA schrittweise auszugleichen. Sie hat derzeit 20 Mitgliedstaaten, die Mehrzahl der EU-Staaten ist an der ESA beteiligt. Die ESA ist gemeinsam mit der NASA Gründungsmitglied des CCSDS. Wir wollen daher auch über zwei Projekte der ESA berichten.

SMART-1 – Start 2003 – Mondumlaufbahn 2004–2006
SMART-1 (**Small Missions for Advanced Research in Technology**, dt. kleine Missionen für fortgeschrittene Technologiestudien) war die erste Raumsonde der ESA, die den Mond erforscht hat. Sie wurde 2003 von Französisch-Guayana gestartet und erreichte erst 2004, d. h. über ein Jahr später, eine Mondumlaufbahn. Diese lange Anreise war der Tatsache geschuldet, dass ein Hauptziel der Mission die Erforschung eines neuartigen, solarelektrisch betriebenen Ionenantriebs war. Auf seiner Umlaufbahn untersuchte SMART-1 anschließend die chemische Zusammensetzung der Mondoberfläche und suchte auch nach Wasser in Form von Eis, bevor die Sonde planmäßig 2006 auf dem Mond aufschlug. Warum erwähnen wir das? Was ist daran so spektakulär, mehr als drei Jahrzehnte nach der ersten bemannten Mondlandung? Nun, ein weiteres Ziel der Mission war auch der Test neuer Navigations- und Kommunikationstechniken. SMART-1 war die erste Raumsonde, die Turbocodes zur Kommunikation nutzte. Allerdings nur testweise, die Sonde war auch sicherheitshalber mit konventioneller Technik ausgestattet. Die größte Herausforderung war dabei natürlich die Decodierung insbesondere bzgl. neu entwickelter Hardwarechips.

Rosetta/Philae – Start 2004 – Umlaufbahn um Komet „Tschuri“ 2014–2016, Philae: Landung auf „Tschuri“ 2014
Das zweite ESA-Projekt, auf das wir eingehen wollen, ist natürlich viel spektakulärer und ging entsprechend ausführlich durch die Presse: **Rosetta, Philae** und der Komet **Tschurjumow-Gerassimenko**. Ein **Komet** (oder **Schweifstern**) ist ein kleiner Himmelskörper von wenigen Kilometern Durchmesser, der in Sonnennähe seiner exzentrischen Bahn einen leuchtenden Schweif entwickelt. Kometen sind in der Frühzeit unseres Sonnensystems entstanden und bestehen aus Eis (Wasser, Kohlendioxid, Methan und Ammoniak), Gestein und Staub. Die meisten der Kometen verschwinden nach Sonnenumrundung in den Bereich jenseits des Planeten Neptun, um nach vielen Jahren wieder in Sonnennähe zurückzukehren. Beim Kometen **Tschurjumow-Gerassimenko** (genannt **„Tschuri“**) ist dies nicht ganz so extrem, er gehört zu den kurzperiodischen Kometen mit einer Umlaufzeit von ca. 6,5 Jahren. Seine kürzeste Entfernung zur Sonne (Perihel) ist derzeit ca. 1,2 AE und seine weiteste (Aphel) ca. 5,7 AE. Zum Vergleich: Der Erdradius ist

nach Definition 1 AE (astronomische Einheit) und der des Jupiters ca. 5,2 AE. Der Komet Tschuri hat die Form einer Kartoffel und misst in alle Richtungen nur ca. 3–4 km.

Die Sonde **Rosetta** wurde 2004 gestartet und schwenkte nach langer Reise – teilweise im *Winterschlaf* – 2014 in eine Umlaufbahn in 10 km Höhe um den Kometen Tschuri ein. Im selben Jahr noch wurde die Landeeinheit **Philae** erfolgreich – obwohl leider etwas verkeilt – auf dem Kometen gelandet. Es war schon eine Sensation, dass man eine Sonde auf einem solch kleinen Himmelskörper in dieser Entfernung absetzen kann. Rosetta und Philae haben den Kometen während seiner aktiven Phase im Jahr 2015 begleitet, in der er in Sonnennähe seinen Schweif ausbildete. Die Forscher erhoffen sich Rückschlüsse auf die chemische und die Isotopenzusammensetzung des frühen Sonnensystems. Auch Rosetta war zur Kommunikation mit der Erde mit einem Turbocode des CCSDS-Standards

Abb. 10.10 Der Zwergplanet Pluto aufgenommen von New Horizons (Quelle NASA [WPNHo])

ausgerüstet. Den Bildern und vorliegenden Ergebnissen nach hat der Code die Erwartungen voll erfüllt.

10.3.4 Anwendung: New Horizons

Start 2006 – Pluto-Passage 2015
Zum Abschluss wollen wir noch über eine Mission der NASA berichten, die wahrlich zu neuen Horizonten aufbrach: **New Horizons**. Es handelt sich dabei um die erste Sonde, die hin zum Zwergplaneten **Pluto** aufbrach. Pluto ist zwischenzeitlich zum Zwergplaneten degradiert, da klar wurde, dass jenseits des letzten Planeten Neptun eine ganze Reihe weiterer relativ kleiner Himmelskörper im sog. **Kuiper-Gürtel** ihre Kreise um die Sonne ziehen (z. B. **Eris**). Zum Vergleich: Die exzentrische, stark geneigte Pluto-Bahn hat Perihel/Aphel zwischen 30 AE und 50 AE, der Neptun hat einen Umlaufradius von ca. 30 AE. Der Pluto besitzt mit einem Durchmesser von ca. 2400 km nur etwa ein Drittel des Volumens des Erdmondes, hat aber seinerseits mit **Charon** (Durchmesser ca. 1200 km) einen vergleichsweise großen Mond.

Die Sonde New Horizons wurde 2006 gestartet und erreichte nach sehr langem Flug 2015 die Pluto-Bahn. Selbst die mit dem besten Weltraumteleskop Hubble gewonnenen Aufnahmen konnten auf Pluto nur bis zu 500 km auflösen, sodass man damit keine Details auf der Oberfläche ausmachen konnte. Dies hat sich nun mit dem New-Horizons-Vorbeiflug geändert. Die Sonde sendete faszinierende Bilder von Pluto und auch Charon zur Erde – wie das Foto von Pluto auf Abb. 10.10 zeigt. Sie ist nun auf ihrem Weg durch den Kuiper-Gürtel, auf dem – wegen des großen Erfolges der Mission – noch weitere Annäherungen an andere größere Objekt geplant sind.

Entgegen des NASA Concatenated Planetary Standard Code hat man sich – unter Berücksichtigung der großen Entfernungen außerhalb des planetarischen Bereichs – bei der New-Horizons-Mission zu einem Turbocode entschieden, und zwar zu dem CCSDS-Standard mit Rate $1/6$. Dieser wird aber sicherheitshalber mit CRC-16-CCITT zur Fehlererkennung zusätzlich geschützt – ebenfalls basierend auf der CCSDS-Empfehlung. Und so fliegt New Horizons weiter und verkündet der *restlichen Welt* von den Errungenschaften der irdischen Informationstechnik ...

Praxisanwendungen im Überblick

Wir haben die zahlreichen Praxisanwendungen fehlererkennender und -korrigierender Codes immer im Zusammenhang mit den gerade behandelten Codes und Codefamilien erläutert. Als Gesamtüberblick sind hier nochmals alle Anwendungen themenbezogen aufgelistet, zusammen mit dem Verweis auf die jeweils relevanten Unterabschnitte.

NASA-Raumfahrt
- Pioneer 9 – Venus (S), Abschn. 9.3.1
- Mariner 6, 7, 9 – Mars (XL), Abschn. 4.3.3, 4.3.4, 9.3.2
- Voyager 1 – Jupiter, Saturn (L), Abschn. 3.4.3, 9.3.3
- Voyager 2 – Jupiter, Saturn, Uranus, Neptun (XL), Abschn. 3.4.3, 5.3.2, 9.3.3, 9.3.4
- Galileo – Jupiter-Umlaufbahn (M), Abschn. 5.3.2, 9.3.6
- Cassini/Huygens – Saturn-Umlaufbahn (M), Abschn. 5.3.2, 9.3.7
- Mars-Rover – von Pathfinder bis Curiosity (M), Abschn. 5.3.3, 9.3.8
- New Horizons – Pluto (M), Abschn. 10.3.4

ESA-Raumfahrt
- SMART-1 – per Turbo zum Mond (S), Abschn. 10.3.3
- Rosetta/Philae – Komet *Tschuri* (M), Abschn. 10.3.3

Kabelgebundene Netzwerktechnologie
- Internet – TCP/IP (M), Abschn. 6.6.7
- LAN – Ethernet (M), Abschn. 6.6.5
- Modbus – Automatisierung (M), Abschn. 6.6.4
- ISDN – digitale Telefonie (S), Abschn. 6.6.6
- DSL – schnelles Internet (L), Abschn. 5.3.8, 9.6.2
- HDLC – High Level Data Link Control (S), Abschn. 6.6.2
- WAN – x.25 (S), Abschn. 6.6.2

Drahtlose Netzwerktechnologie
- WLAN – lokales Funknetz (S), Abschn. 6.6.3
- WiMAX – regionales Funknetz (M), Abschn. 8.3.4

O. Manz, *Fehlerkorrigierende Codes*, DOI 10.1007/978-3-658-14652-8

Schnittstellen
- USB – Universal Serial Bus (M), Abschn. 6.6.3
- Bluetooth – Funkschnittstelle (M), Abschn. 6.6.3
- SAE-J1850 – On-Board-Diagnose (S), Abschn. 6.6.3

Mobilfunk
- GSM – 2. Generation (XL), Abschn. 9.6.3
- UMTS – 3. Generation (L), Abschn. 9.6.3, 10.3.1
- LTE – 4. Generation (L), Abschn. 9.6.3, 10.3.1

Sonstige Funktechnik
- Pager – Personenrufempfänger (M), Abschn. 7.2.5
- ALE – Automatic Link Establishment (S), Abschn. 3.4.4
- MIDS – NATO-Militärfunk (S), Abschn. 5.3.6

Digitalfernsehen
- DVB-S/T/C – 1. Generation (L), Abschn. 5.3.7, 8.3.1, 8.3.2, 9.6.1
- DVB-S2/T2/C2 – 2. Generation (XL), Abschn. 8.3.1, 8.3.3

Satellitennavigation
- GPS – zivile L1/L2/L5-Bänder (XL), Abschn. 3.2.6, 8.1.6, 9.6.4
- Galileo – Europa – E1-Band (S), Abschn. 9.6.4

Speichermedien
- Arbeitsspeicher – ECC-Memories (M), Abschn. 3.2.5
- Festplatte – magnetischer Speicher (L), Abschn. 5.5.5, 8.1.5
- Audio-CD – optischer (Musik-)Speicher (XL), Abschn. 5.5.1, 5.5.2, 5.5.3
- Daten-DVD – optischer Datenspeicher (L), Abschn. 5.5.4
- Speicherchip – USB-Stick, SD-Karten (M), Abschn. 6.6.3, 6.6.9
- Zip – Datenkompression (S), Abschn. 6.6.10

Barcodes
- QR – 2-D-Barcode (XL), Abschn. 5.3.4, 7.3.1–7.3.4
- Aztec – 2-D-Barcode (M), Abschn. 5.3.4
- DataMatrix – 2-D-Barcode (S), Abschn. 5.3.4
- PDF 417 – gestapelter Barcode (L), Abschn. 5.3.5, 7.1.4

Artikel
- EAN – Artikelnummer mit Barcode (L), Abschn. 1.4.3
- ISBN – Buchnummer (M), Abschn. 1.2.4, 1.4.3

Rund ums Geld
- Euro – Seriennummer (M), Abschn. 1.2.6
- ISIN – Wertpapierkennnummer (L), Abschn. 1.2.5
- IBAN – Bankkontonummer (L), Abschn. 1.4.4

Kryptografie
- Stromchiffre – Vernam (M), Abschn. 7.5.3
- PGP – Pretty Good Privacy (M), Abschn. 6.6.8
- McEliece-Kryptosystem – Goppa-Codes (S), Abschn. 7.6.5

Medizin
- Diagnostik – fotoakustische Bildgebung (S), Abschn. 3.4.5

Sportwetten
- ODDSET – Fußballsiegwette (M), Abschn. 3.4.2
- Veikkaaja – finnisches Fußballmagazin (S), Abschn. 3.4.1

Legende
S kurz umrissen
M das Wesentliche erläutert
L ausführlich dargestellt
XL im Detail beschrieben

Literatur

[Abr] Abrantes, S.: From BCJR to Turbo Decoding: MAP Algorithms made Easier (Übersichtsaufsatz). http://paginas.fe.up.pt/~sam/textos/From%20BCJR%20to%20turbo.pdf Porto /Portugal, Lawrence/KS-USA (2004)

[ANSI] American National Standard Institute ANSI: Network and Customer Installation Interfaces – Asymmetric Digital Subscriber Line (ADSL) Metallic Interface (Standardspezifikation). http://ftp.tiaonline.org/TR-30/TR-30.3/Public/WH-027.pdf Hawaii/USA (1998)

[BaS] Balatsoukas-Stimming, A.: Repeat-Accumulate Codes (Kursfolien). http://www.telecom.tuc.gr/~alex/lectures/lecture7.pdf Kreta/Griechenland (2009)

[Bau] Bauer, C.: Modbus-Grundlagen (Technischer Überblick). https://www.gossenmetrawatt.com/resources/me/sineax_cam/modbus_basics_d.pdf Wohlen/Schweiz (2006)

[BNS] Beutelspacher, A., Neumann, H., Schwarzpaul, Th.: Kryptografie in Theorie und Praxis (Lehrbuch). Vieweg+Teubner, Wiesbaden/Deutschland (2010)

[BSW] Beutelspacher, A., Schwenk, A., Wolfenstetter, K.-D.: Moderne Verfahren der Kryptographie (Fachbuch). Springer Vieweg, Heidelberg/Deutschland (2006)

[Bla] Blahut, R.: Theory and Practice of Error Control Codes (Lehrbuch). Addison-Wesley, Reading/MA-USA (1983)

[Blö] Blömer, J.: Algorithmische Codierungstheorie II (Vorlesungsskript). http://www-old.cs.uni-paderborn.de/fileadmin/Informatik/AG-Bloemer/lehre/2009/ss/ac2/skript.pdf Paderborn/Deutschland (2009)

[Bre] Brenner, P.: IEEE 802.11 Protocol (Technisches Tutorium). http://www.sss-mag.com/pdf/802_11tut.pdf BreezeCOM/Alvarion, TelAviv/Israel (1996)

[Byu] Byun, J.: 3GPP2/3GPP Coding: Turbo Code and Turbo Interleaver Summary (Übersichtsaufsatz). http://web.stanford.edu/class/ee379b/class_reader/3G_turbocode_interleaver.pdf Stanford/CA-USA (2004)

[CDFST] Callas, J., Donnerhacke, L., Finney, H., Shaw, D., Thayer, R.: OpenPGP Message Format (Internetinformation). https://tools.ietf.org/html/rfc4880 Network Working Group (2007, geöffnet 2016)

[Cha] Charlesworth, P.B.: Turbo Codes (Übersichtsaufsatz). http://www.philsrockets.org.uk/turbocodes.pdf philsrockets.org/Großbritannien (2000)

[Che] Cherowitzo, W.: Reed-Muller-Codes (Kursfolien). http://www-math.ucdenver.edu/~wcherowi/courses/m7823/reedmuller.pdf Denver/CO-USA (2009)

[CCS1] Consultative Committee for Space Data Systems CCSDS: TM Synchronization and Channel Coding (Blue Book). http://public.ccsds.org/publications/archive/131x0b2ec1.pdf Washington/DC-USA (2011)

[CCS2] Consultative Committee for Space Data Systems CCSDS: TM Space Data Link Protocol (Blue Book). http://public.ccsds.org/publications/archive/132x0b2.pdf Washington/DC-USA (2015)

[DAm] D'Amours, C.: Error Control Coding (Kursfolien). http://www.site.uottawa.ca/~damours/courses/ELG_5372/ Ottawa/Kanada (2007)

[DBo] DeBoy, C.C. (et al): The RF Telecommunications System for the New Horizons Mission to Pluto (Übersichtsaufsatz, 55th International Astronautical Congress). http://www.uhf-satcom.com/amateurdsn/Paper-969.pdf Baltimore/MD-USA, Vancouver/Kanada (2004)

[Din] Dingel, J.: Irregular Repeat Accumulate- and Low-Density-Parity-Check-Codes (Präsentationsfolien). http://www14.in.tum.de/konferenzen/Jass05/courses/4/presentations/Janis%20Dingel-Low%20Density%20Parity%20Check%20Codes.pdf St. Petersburg/Russland, München/Deutschland (2005)

[DJE] Divsalar, D., Jin, H., McEliece, R.: Coding Theorems for "Turbo-Like" Codes (Forschungsartikel). http://www.systems.caltech.edu/systems/rjm/papers/Allerton98.pdf Pasadena/CA-USA (1998)

[Eiw] Eiwen, D.: Die Golay Codes (Diplomarbeit). http://homepage.univie.ac.at/daniel.eiwen/Eiwen_Golay.pdf Wien/Österreich (2008)

[Emo] Emotive: SAE J1850 (Internetinformation) https://www.emotive.de/doc/car-diagnostic-systems/bus-systems/sae-j1850 Stuttgart/Deutschland (2011)

[ETDVB] European Telecommunications Standards Institute ETSI: Digital Video Broadcasting (DVB); Frame Structure Channel Coding and Modulation (Standardspezifikation). http://www.etsi.org/deliver/etsi_en/302700_302799/302769/01.01.01_60/en_302769v010101p.pdf Sophia Antipolis/Frankreich (2010)

[ETGSM] European Telecommunications Standards Institute ETSI: GSM Digital Cellular Telecommunications System; Channel Coding (Technische Spezifikation). http://www.etsi.org/deliver/etsi_gts/05/0503/05.00.00_60/gsmts_0503v050000p.pdf Sophia Antipolis/Frankreich (1996)

[ETISD] European Telecommunications Standards Institute ETSI: Broadband Integrated Services Digital Network (ISDN); ISDN ATM Layer Specification (Funktionale Spezifikation). http://www.etsi.org/deliver/etsi_i_ets/300200_300299/30029802/01_60/ets_30029802e01p.pdf Sophia Antipolis/Frankreich (1995)

[FCT] FCT Faculdade de Ciencias e Tecnologia: Channel Codification in GSM and UMTS Systems (Übersichtsaufsatz). http://tele1.dee.fct.unl.pt/csf_2012_2013/laboratorio/TRAb_SCW_coding_2012_2013_eng.pdf Lissabon/Portugal (2012)

[Fer] Fernandes, G.: Parallel Algorithms and Architectures for LDPC Decoding (Dissertation). https://estudogeral.sib.uc.pt/bitstream/10316/14583/3/Tese_Gabriel%20Fernandes.pdf Coimbra/Portugal (2010)

[Frak] Frank, H.: Einführung in Ethernet (Vorlesungsskript). https://www.hs-heilbronn.de/1749557/Ethernet Heilbronn/Deutschland (2010)

[Frae] Franke, D.: IBAN International Bank Account Number (Internetinformation) http://www.iban.de/ Leipzig/Deutschland (geöffnet 2016)

[For] Forney, D.: The Viterbi Algorithm (Forschungsartikel). http://www.systems.caltech.edu/EE/Courses/EE127/EE127A/handout/ForneyViterbi.pdf Proc. IEEE 61, Newton/MA-USA (1973)

[FTDI] Future Technology Devices International Limited FTDI: USB Data Packet Structure (Technischer Überblick). http://www.ftdichip.com/Support/Documents/TechnicalNotes/TN_116_USB%20Data%20Structure.pdf Glasgow/Großbritannien (2009)

[Gal] Gallager, R.: Low-Density Parity-Check Codes (Dissertation). http://www.rle.mit.edu/rgallager/documents/ldpc.pdf Cambridge/MA-USA (1963)

[Gar] Garcia-Pena. A.: GNSS Navigation Message Analysis and Perspectives (Präsentationsfolien). http://cct.cnes.fr/system/files/cnes_cct/928-pds/public/030_Animations/2014/Symposium/Symposium_Axel_Garcia_v5.pdf ENAC Toulouse/Frankreich (2014)

[GzPDF] Grandzebu: The PDF 417 Code (Internetinformation). http://grandzebu.net/informatique/codbar-en/pdf417.htm (geöffnet 2016)

[Gol] Golay, M.: Notes on Digital Coding (Forschungsartikel). http://www.maths.manchester.ac.uk/~ybazlov/code/golay_paper.pdf Proc. IRE 37, Fort Monmouth/NJ-USA (1949)

[HCV] Habinc, S., Calzolari, G., Vassallo, E.: Development Plan for Turbo Encoder Core and Devices Implementation Updated – CCSDS Telemetry Channel Coding Standard (Übersichtsaufsatz). http://microelectronics.esa.int/vhdl/doc/TurboCCSDS.pdf Darmstadt/Deutschland, Noordwijk/Niederlande (1998)

[Ham] Hamming, R.: Error Detecting and Error Correcting Codes (Forschungsartikel). Bell Syst. Tech. J. 26, New York/NY-USA (1950)

[Hau1] Hauck, P.: Codierungstheorie (Vorlesungsskript). http://dm.inf.uni-tuebingen.de/skripte/Codierungstheorie/CodierungstheorieWS0506.pdf Tübingen/Deutschland (2005)

[Hau2] Hauck, P.: Codierungstheorie (Vorlesungsskript). http://dm.inf.uni-tuebingen.de/skripte/Codierungstheorie/CodierungstheorieSS2010.pdf Tübingen/Deutschland (2010)

[HGST] HGST Western Digital Company: Iterative Detection Read Channel Technology in Hard Disk Drives (White Paper). https://www.hgst.com/sites/default/files/resources/IDRC_WP_final.pdf San Jose/CA-USA (2008)

[Hol] Holtkamp, H.: Einführung in Bluetooth (Seminararbeit). http://www.rvs.uni-bielefeld.de/lecture/bluetooth/bluetooth.pdf Bielefeld/Deutschland (2003)

[HBH] Hüttinger S., ten Brink, S., Huber, J.: Turbo-Code Representation of RA-Codes and DRS-Codes for Reduced Decoding Complexity (Forschungsartikel, Conf. Inform. Sciences & Systems). http://www.lit.eei.uni-erlangen.de/papers/Ciss01_simon_huettinger.pdf Erlangen/Deutschland, Holmdel/NJ-USA (2001)

[JaTo] Jankvist, U., Toldbod, B.: Mars Exploration Rover – Mathematics and People behind the Mission (Übersichtsaufsatz). http://scholarworks.umt.edu/cgi/viewcontent.cgi?article=1068&context=tme The Montana Math. Enth. 4, Roskilde/Dänemark (2007)

[Joo] Joost, M.: Turbo-Codes (Übersichtsaufsatz). http://www.michael-joost.de/turbo.pdf Krefeld/Deutschland (2010)

[Käs] Käsper, E.: Turbo-Codes (Übersichtsaufsatz). http://www.mif.vu.lt/~skersys/vsd/turbo/kasper-turbo.pdf Helsinki/Finnland (2005)

[KeWa] Kersting, G., Wakolbinger, A.: Elementare Stochastik (Lehrbuch). Birkhäuser-Springer, Basel/Schweiz (2010)

[Kla] Klaas, L.: Informationstheorie und Codierung (Vorlesungsskript). http://www.fh-bingen.de/fileadmin/user_upload/Lehrende/Klaas_Lothar/Skripte/INCO.pdf Bingen/Deutschland (2015)

[Kod] Kodnar, J.: Bedeutung der EURO-Kontrollnummer (Internetinformation). http://www.geldschein.at/euro_seriennummer.html Wien/Österreich (geöffnet 2016)

[KüWü] Kühn, V, Wübben, D.: Kanalcodierung I (Vorlesungsskript). http://www.ant.uni-bremen.de/sixcms/media.php/102/9490/kc1_top.pdf Bremen/Deutschland (2016)

[Lei] Leinen, F.: Codierungstheorie am Beispiel von Musik-CDs (Lehrerfortbildung). http://www.alt.mathematik.uni-mainz.de/schule/lehrkraefte/lehrkraeftefortbildungen/vor_2007/2006.2/cd-codierung-2.pdf Mainz/Deutschland (2006)

[LNT] LNT Lehrstuhl für Nachrichtentechnik: Beispiele von Nachrichtensystemen (Lerntutorial). http://www.lntwww.de/Beispiele_von_Nachrichtensystemen/Kapitel162.html München/Deutschland (2015)

[Mas] Massey, J.: Deep-Space Communications and Coding – A Marriage Made in Heaven (Übersichtsaufsatz). http://www.isiweb.ee.ethz.ch/archive/massey_pub/pdf/BI321.pdf Contr. Inform. Sc. 182, Heidelberg/Deutschland (1992)

[Mat] Matzat, B.H.: Codierungstheorie (Vorlesungsskript). http://www.iwr.uni-heidelberg.de/~Heinrich.Matzat/PDF/Codierungstheorie.pdf Heidelberg/Deutschland (2007)

[Max] Maxein, S.: Benutzen einer SD-Speicherkarte mit dem ATmega-Microcontroller (Übersichtsaufsatz). http://www.uni-koblenz.de/~physik/informatik/ECC/sd.pdf Koblenz/Deutschland (2008)

[Mie] Mienkina, M.: Kodierte Anregung in der Photoakustischen Bildgebung (Dissertation). http://www-brs.ub.ruhr-uni-bochum.de/netahtml/HSS/Diss/MienkinaMartinP/diss.pdf Bochum/Deutschland (2010)

[Mow] Mow, W.H.: Burst Error Correction (Kursfolien). http://course.ee.ust.hk/elec332/332T7.pdf Hongkong (2011)

[Mul] Muller, D.: Application of Boolean Algebra to Switching Circuit Design and to Error Detection (Forschungsartikel). IEEE Trans. 3, Washington/DC-USA (1954)

[OCC] Onyszchuk, I., Cheung, K.-M., Collins, O.: Quantization Loss in Convolutional Decoding (Forschungsartikel). http://www.ee.ust.hk/~eejinjie/research_files/viterbi_files/Quantization%20loss%20in%20convolutional%20decoding.pdf IEEE Trans. Comm. 41, New York/NY-USA (1993)

[PAG] Pique, J., Almazan, I., Garcia, D.: Bluetooth (Übersichtsaufsatz). http://web.udl.es/usuaris/carlesm/docencia/xc1/Treballs/Bluetooth.Treball.pdf Lleida/Spanien (2001)

[ReSo] Reed, I., Solomon, G.: Polynomial Codes over certain Finite Fields (Forschungsartikel). SIAM J. 8, Philadelphia/PA-USA (1960)

[Rei] Reimers, U.: DVB Digitale Fernsehtechnik – Datenkompression und Übertragung (Fachbuch). Springer, Heidelberg/Deutschland (2008)

[Röd] Röder, M.: Effiziente Viterbi Decodierung und Anwendung auf die Bildübertragung in gestörten Kanälen (Diplomarbeit). http://lips.informatik.uni-leipzig.de/files/2001-48.pdf Leipzig/Deutschland (2001)

[Ros1] Rosnes, E.: Error Control Coding, Lecture 11 MAP Decoding – BCJR Algorithm (Kursfolien). http://www.ii.uib.no/~eirik/INF244/Lectures/Lecture11.pdf Bergen/Norwegen (2006)

[Ros2] Rosnes, E.: Error Control Coding, Lecture 14 Turbo Codes (Kursfolien). http://www.ii.uib.no/~eirik/INF244/Lectures/Lecture14.pdf Bergen/Norwegen (2006)

[Rud] Rudolph, D.: Fehlerschutz-Codierung 2 – Faltungscode (Vorlesungsskript). http://www.diru-beze.de/funksysteme/skripte/DiFuSy/DiFuSy_FaltungsCode_WS0405.pdf Berlin/Deutschland (2004)

[SaD] SanDisk: SanDisk Secure Digital Card (Product Manual). http://www.convict.lu/pdf/ProdManualSDCardv1.9.pdf Sunnyvale/CA-USA (2003)

[SaG] Sauer-Greff, W.: Einführung in die Informations- und Codierungstheorie – II Codierungstheorie (Vorlesungsskript). http://nt.eit.uni-kl.de/fileadmin/lehre/ict/skript/codier.pdf Kaiserslautern/Deutschland (2012)

[VCS] Valenti, M., Cheng, S., Seshadri, R.: Digital Video Broadcasting (Forschungsartikel). http://www.csee.wvu.edu/~mvalenti/documents/DVBChapter.pdf Morgantown/WV-USA (2004)

[vLi82] van Lint, J.: Introduction to Coding Theory (Lehrbuch). Springer, Heidelberg/Deutschland (1982)

[vLi99] van Lint, J.: Introduction to Coding Theory (Lehrbuch). Springer, Heidelberg/Deutschland (1999)

[vLvG] van Lint, J., van der Geer, G.: Introduction to Coding Theory and Algebraic Geometry (Tagungsband). DMV-Seminar 12, Birkhäuser, Basel/Schweiz (1988)

[WPPlS] Wikipedia: Planetensystem (Internetenzyklopädie). https://de.wikipedia.org/wiki/Planetensystem (geöffnet 2016)

[WPPio] Wikipedia: Pioneer – Raumsonden-Programm (Internetenzyklopädie). https://de.wikipedia.org/wiki/Pioneer_(Raumsonden-Programm) (geöffnet 2016)

[WPMar] Wikipedia: Mariner (Internetenzyklopädie). https://de.wikipedia.org/wiki/Mariner (geöffnet 2016)

[WPVo1] Wikipedia: Voyager 1 (Internetenzyklopädie). https://de.wikipedia.org/wiki/Voyager_1 (geöffnet 2016)

[WPVo2] Wikipedia: Voyager 2 (Internetenzyklopädie). https://de.wikipedia.org/wiki/Voyager_2 (geöffnet 2016)

[WPGal] Wikipedia: Galileo–Raumsonde (Internetenzyklopädie). https://de.wikipedia.org/wiki/Galileo_(Raumsonde) (geöffnet 2016)

[WPCaH] Wikipedia: Cassini-Huygens (Internetenzyklopädie). https://de.wikipedia.org/wiki/Cassini-Huygens (geöffnet 2016)

[WPMPa] Wikipedia: Mars-Pathfinder (Internetenzyklopädie). https://de.wikipedia.org/wiki/Mars_Pathfinder (geöffnet 2016)

[WPMSO] Wikipedia: Mars-Exploration-Rover (Internetenzyklopädie). https://de.wikipedia.org/wiki/Mars_Exploration_Rover (geöffnet 2016)

[WPMCu] Wikipedia: Mars Science Laboratory (Internetenzyklopädie). https://de.wikipedia.org/wiki/Mars_Science_Laboratory (geöffnet 2016)

[WPNHo] Wikipedia: New Horizons (Internetenzyklopädie). https://de.wikipedia.org/wiki/New_Horizons (geöffnet 2016)

[WPESA] Wikipedia: Europäische Weltraumorganisation (Internetenzyklopädie). https://de.wikipedia.org/wiki/Europ%C3%A4ische_Weltraumorganisation (geöffnet 2016)

[WPSm1] Wikipedia: SMART-1 (Internetenzyklopädie). https://de.wikipedia.org/wiki/SMART-1 (geöffnet 2016)

[WPRoP] Wikipedia: Rosetta-Sonde (Internetenzyklopädie). https://de.wikipedia.org/wiki/Rosetta_(Sonde) (geöffnet 2016)

[WPECC] Wikipedia: Speichermodul (Internetenzyklopädie). https://de.wikipedia.org/wiki/Speichermodul (geöffnet 2016)

[WPCD] Wikipedia: Compact Disk (Internetenzyklopädie). https://de.wikipedia.org/wiki/Compact_Disc (geöffnet 2016)

[WPDVD] Wikipedia: DVD (Internetenzyklopädie). https://de.wikipedia.org/wiki/DVD (geöffnet 2016)

[WPFLW] Wikipedia: Festplattenlaufwerk (Internetenzyklopädie). https://de.wikipedia.org/wiki/Festplattenlaufwerk (geöffnet 2016)

[WPSDK] Wikipedia: SD-Karte (Internet-Enzyklopädie). https://de.wikipedia.org/wiki/SD-Karte (geöffnet 2016)

[WPZIP] Wikipedia: ZIP – File Format (Internetenzyklopädie). https://en.wikipedia.org/wiki/Zip_(file_format) (geöffnet 2016)

[WPAzt] Wikipedia: Aztec-Code (Internetenzyklopädie). https://de.wikipedia.org/wiki/Aztec-Code (geöffnet 2016)

[WPPDF] Wikipedia: PDF417 (Internetenzyklopädie). https://de.wikipedia.org/wiki/PDF417 (geöffnet 2016)

[WPQR] Wikipedia: QR-Code (Internetenzyklopädie). https://de.wikipedia.org/wiki/QR-Code (geöffnet 2016)

[WVQR1] Wikiversity: Reed–Solomon Codes for Coders (Internetuniversität). https://en.wikiversity.org/wiki/Reed%E2%80%93Solomon_codes_for_coders (geöffnet 2016)

[WVQR2] Wikiversity: Reed–Solomon Codes for Coders/Additional Information (Internetuniversität). https://en.wikiversity.org/wiki/Reed%E2%80%93Solomon_codes_for_coders/Additional_information (geöffnet 2016)

[WPOSI] Wikipedia: OSI-Modell (Internetenzyklopädie). https://de.wikipedia.org/wiki/OSI-Modell#Schicht_2_.E2.80.93_Sicherungsschicht (geöffnet 2016)

[WPEtN] Wikipedia: Ethernet (Internetenzyklopädie). https://de.wikipedia.org/wiki/Ethernet#FCS_.28Frame_Check_Sequence.29 (geöffnet 2016)

[WPInN] Wikipedia: Internetprotokollfamilie (Internetenzyklopädie). https://de.wikipedia.org/wiki/Internetprotokollfamilie (geöffnet 2016)

[WPIP] Wikipedia: IP-Paket (Internet-Enzyklopädie). https://de.wikipedia.org/wiki/IP-Paket (geöffnet 2016)

[WPTCP] Wikipedia: Transmission Control Protocol (Internetenzyklopädie). https://de.wikipedia.org/wiki/Transmission_Control_Protocol (geöffnet 2016)

[WPISD] Wikipedia: Integrated Services Digital Network (Internetenzyklopädie). https://de.wikipedia.org/wiki/Integrated_Services_Digital_Network (geöffnet 2016)

[WPDSL] Wikipedia: Digital Subscriber Line (Internetenzyklopädie). https://de.wikipedia.org/wiki/Digital_Subscriber_Line (geöffnet 2016)

[WPALE] Wikipedia: Automatic Link Establishment (Internetenzyklopädie). https://de.wikipedia.org/wiki/Automatic_Link_Establishment (geöffnet 2016)

[WPMID] Wikipedia: Multifunctional Information Distribution System (Internetenzyklopädie). https://de.wikipedia.org/wiki/Multifunctional_Information_Distribution_System (geöffnet 2016)

[WPPag] Wikipedia: Funkmeldeempfänger (Internetenzyklopädie). https://de.wikipedia.org/wiki/Funkmeldeempf%C3%A4nger (geöffnet 2016)

[WPDVB] Wikipedia: Digital Video Broadcasting (Internetenzyklopädie). https://de.wikipedia.org/wiki/Digital_Video_Broadcasting (geöffnet 2016)

[WPGPS] Wikipedia: GPS Signals (Internetenzyklopädie). https://en.wikipedia.org/wiki/GPS_signals (geöffnet 2016)

[WPGSM] Wikipedia: Global System for Mobile Communications (Internetenzyklopädie). https://de.wikipedia.org/wiki/Global_System_for_Mobile_Communications (geöffnet 2016)

[WPUMT] Wikipedia: Universal Mobile Telecommunications System (Internetenzyklopädie). https://de.wikipedia.org/wiki/Universal_Mobile_Telecommunications_System (geöffnet 2016)

[WPLTE] Wikipedia: Long Term Evolution (Internetenzyklopädie). https://de.wikipedia.org/wiki/Long_Term_Evolution (geöffnet 2016)

[WPEur] Wikipedia: Eurobanknoten (Internetenzyklopädie). https://de.wikipedia.org/wiki/Eurobanknoten#Nummerierungssysteme_2 (geöffnet 2016)

[WPISI] Wikipedia: International Securities Identification Number (Internetenzyklopädie). https://de.wikipedia.org/wiki/International_Securities_Identification_Number (geöffnet 2016)

[WPIBA] Wikipedia: IBAN (Internetenzyklopädie). https://de.wikipedia.org/wiki/IBAN#Berechnung_der_Pr.C3.BCfsumme (geöffnet 2016)

[WPEAN] Wikipedia: European Article Number (Internetenzyklopädie). https://de.wikipedia.org/wiki/European_Article_Number (geöffnet 2016)

[WPCRC] Wikipedia: Zyklische Redundanzprüfung (Internetenzyklopädie). https://de.wikipedia.org/wiki/Zyklische_Redundanzpr%C3%BCfung (geöffnet 2016)

[WPCPo] Wikipedia: Polynomial Representations of Cyclic Redundancy Checks (Internetenzyklopädie). https://en.wikipedia.org/wiki/Polynomial_representations_of_cyclic_redundancy_checks (geöffnet 2016)

[WPPGP] Wikipedia: Base64 (Internetenzyklopädie). https://de.wikipedia.org/wiki/Base64#Radix-64 (geöffnet 2016)

[WPBM] Wikipedia: Berlekamp-Massey-Algorithmus (Internetenzyklopädie). https://de.wikipedia.org/wiki/Berlekamp-Massey-Algorithmus (geöffnet 2016)

[WPChi] Wikipedia: Chien Search (Internetenzyklopädie). https://en.wikipedia.org/wiki/Chien_search (geöffnet 2016)

[WPGol] Wikipedia: Golay-Code (Internetenzyklopädie). https://de.wikipedia.org/wiki/Golay-Code (geöffnet 2016)

[Wil] Willems, W.: Codierungstheorie (Lehrbuch). De Gruyter, Berlin/Deutschland (1999)

[Wil2] Willems, W.: Codierungstheorie und Kryptographie (Lehrbuch). Birkhäuser-Springer, Basel/Schweiz (2008)

[Wol] Wolf, J.K.: An Introduction to Error Correcting Codes Part 2 (Kursfolien). http://circuit.ucsd.edu/~yhk/ece154c-spr15/ErrorCorrectionII.pdf San Diego/CA-USA (2010)

[Zar] Zarei, S.: LTE – Der Mobilfunk der Zukunft – Channel Coding and Link Adaptation (Seminararbeit). http://www.lmk.lnt.de/fileadmin/Lehre/Seminar09/Ausarbeitungen/Ausarbeitung_Zarei.pdf Erlangen/Deutschland (2009)

[Zha] Zhao, X.: Research on LDPC Decoder Design Methodology for Error-Correcting Performance and Energy-Efficiency (Dissertation). https://dspace.wul.waseda.ac.jp/dspace/bitstream/2065/44770/3/Honbun-6584.pdf Tokyo/Japan (2014)

[Zit] Zitterbart, M.: Kommunikation und Datenhaltung; 3. Protokollmechanismen (Kursfolien). http://www.ipd.uni-karlsruhe.de/~komdat/files/kom/kap03_protokollmechanismen_notes.pdf Karlsruhe/Deutschland (2009)

Sachverzeichnis