Fehlerkorrektur bei digitalen Nachrichten am Beispiel des Reed-Solomon-Codes

Benjamin Antesberger

10. November 2025

Inhaltsverzeichnis

1 Einleitung **3**

2 Mathematische Grundlagen **3**

2.1 Modulare Arithmetik 3

2.2 Lagrange Interpolation 4

2.3 Galoiskörper 4

3 Der Algorithmus **5**

3.1 Die Grundidee 8

3.2 Die Vorbereitung 9

3.3 Die Verschlüsselung 11

3.4 Die Entschlüsselung 13

3.4.1 Schritt 1: Die erste Interpolation 13

3.4.2 Schritt 2: Die Fehlerfindung 14

3.4.3 Schritt 3: Die zweite Interpolation 16

3.4.4 Grenzen unseres Codes 17

4 Zusammenfassung **18**

A Restlicher Programmcode **19**

A.1 Helferfunktionen 19

A.2 Die Klassen „gf_element“ und „poly“ 19

A.3 Beispielcode für die Kodierung 24

1 Einleitung

Heutzutage wird der größte Teil aller Nachrichten auf digitalem Weg versandt. Der Begriff „Nachricht“ wird hierbei sehr weit gefasst. So gilt zum Beispiel eine Textnachricht über Funk aber auch der Datentransfer von einer CD auf den Laptop oder generell jede Art von digitalem Datenaustausch als eine Nachricht. Bei der Übertragung von Nachrichten auf digitalem Weg – z.B. beim Einlesen einer CD – kann es durch externe Einflüsse – im Falle der CD zum Beispiel durch Staubkörner – immer wieder zu Fehlern bei der Datenübertragung kommen. Natürlich kann man einfach mehrere Exemplare einer Nachricht versenden und dann die Version nehmen, bei der sich die meisten Exemplare einig sind. Das ist jedoch sehr ineffizient, weil selbst mit der geringsten Redundanz bereits zwei Drittel der versendeten Daten keine Information liefern.[1] Eine derartige Absicherung vor Übertragungsfehlern kostet somit in erster Linie Bandbreite, die zum Beispiel im Falle des Internets von anderen hätte benutzt werden können, aber auch Rechenleistung, Zeit und letzendlich auch Geld.
Um nun aber Fehler erkennen und ggf. korrigieren zu können, ohne dass zu viel Platz in der Nachricht für diese Korrektur ‚verschwendet‘ werden muss, gibt es diverse Methoden von welchen im Folgenden eine – die Reed-Solomon-Codes (RS-Codes) – näher beleuchtet und in einem Beispielprogramm implementiert wird.

Dieser Aufsatz stützt sich primär auf das Buch „Konkrete Mathematik (nicht nur) für Informatiker“ von Edmumd Weitz.[2] Aus Gründen der Komplexität wurden einzelne Schritte des dem Buch entnommenen Vorgehens durch einfacher zu verstehende Vorgänge mit selbem Resultat ersetzt. Der Programmcode wurde jedoch – bis auf den Algorithmus zur Bestimmung des kgV und einen Codeschnipsel zum Test für ein irreduzibles Polynom – vollständig selbst geschrieben und aus den mathematischen Konzepten erarbeitet.

2 Mathematische Grundlagen

2.1 Modulare Arithmetik

Die Modulo-Operation nimmt als Input zwei Zahlen $a \in \mathbb{Z}$ und $b \in \mathbb{Z}$ und gibt als output den Rest, der bei der Division von a mit b übrig bleibt.[3]

[1] Weitz 2021, S. 770.
[2] Weitz 2021.
[3] Weitz 2021, S. 34.

So gilt zum Beispiel folgende Gleichung:

$$3 \equiv 1 \pmod 2$$

Rechnet man Modulo n, so sagt man auch, dass man in $\mathbb{Z}/n\mathbb{Z}$ rechnet. $\mathbb{Z}/n\mathbb{Z}$ ist die Menge aller Reste die bei der Division mit n entstehen können.[4] Im Folgenden wird $\mathbb{Z}/n\mathbb{Z}$ mit $\mathbb{Z}_n$ bezeichnet, wenn n eine Primzahl ist.[5]

2.2 Lagrange Interpolation

Die Lagrange Interpolation ist ein Weg, das Polynom, das durch eine Menge an gegebenen Punkten läuft, zu bestimmen.
Die Lagrange-Interpolation lässt sich folgendermaßen kurz zusammenfassen:
Es sei $m(x)$ ein Polynom von Grad n und $\{(x_i, y_i) | 0 \leq i \leq n\}$ eine Menge mit $n+1$ verschiedenen Werten y_i von $m(x)$ an den jeweiligen Stellen x_i.
$m(x)$ lässt sich nun folgendermaßen bestimmen:[6][7]

$$m(x) = \sum_{i=0}^{n} y_i l_i \tag{1}$$

mit

$$l_i = \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{(x - x_j)}{(x_i - x_j)} \tag{2}$$

2.3 Galoiskörper

Ein **Körper** $(F, +, \cdot)$ ist eine Menge F, auf welcher zwei binäre Operationen $+, \cdot : F \times F \to F$[8] definiert sind. Diese Operationen nennen wir Addition bzw. Multiplikation. Für Addition und Multiplikation gelten folgende Gesetze[9]:

Abgeschlossenheit: $\forall a, b \in F : (a + b \in F,\ a \cdot b \in F)$

Jede Addition bzw. Multiplikation über F liefert als Ergebnis immer Elemente von F.

Assoziativität: $\forall a, b, c \in F : [(a + b) + c = a + (b + c)] \wedge [(a \cdot b) \cdot c = a \cdot (b \cdot c)]$

[4] Weitz 2021, S. 38.

[5] Schreibweise übernommen von Weitz 2021, S. 68

[6] Deford 2015, S. 2.

[7] Kværnø 2021, S. 3.

[8] zu verstehen als: „Die Operation nimmt zwei Elemente aus F als Input und gibt ein Element aus F als Output zurück.“, daher auch die Bezeichnung „binär“.

[9] Madritsch 2010, S. 5–6.

Kommutativität: $\forall a,b \in F : (a+b=b+a) \wedge (a \cdot b = b \cdot a)$

Additive Identität: $\exists! x \in F : \forall a \in F : a+x=a$ Dieses x bezeichnen wir mit 0.[10]

Multiplikative Identität: $\exists! x \in F : \forall a \in F : a \cdot x = a$ Dieses x bezeichnen wir mit 1.[11]

Additives Inverses: $\forall a \in F : \exists x \in F : a+x=0$ Dieses x nennen wir das additive Inverse von a oder kurz $-a$.

Multiplikatives Inverses: $\forall a \in F \backslash \{0\} : \exists x \in F : a \cdot x = 1$ Dieses x nennen wir das multiplikative Inverse von a oder kurz a^{-1}.

Wie genau man in einem Körper addiert oder multipliziert, ist dabei nebensächlich. Wichtig ist, dass die Operationen die oben genannten Gesetzmäßigkeiten erfüllen. Zum Beispiel ist $(\mathbb{R},+,\cdot)$ ein Körper, weil er diese Bedingungen erfüllt.

Ein **endlicher Körper**, auch **Galoiskörper** genannt, ist nun ein Körper, der endlich viele Elemente enthält.[12] Eine *primitive Einheitswurzel* α ist ein Element, dessen Potenzen alle anderen Elemente des Körpers – außer die 0 – ergeben. Es gilt also $\{\alpha^k | 0 \leq k < 255\} = GF(2^8)\backslash\{0\}$.[13]

Ist p eine Primzahl, dann ist $\mathbb{Z}_p$ ein Körper.[14] Allgemein gibt es für jede Primzahl p und jede positive ganze Zahl n – bis auf Isomorphie[15] – genau einen Körper mit p^n Elementen.[16][17].

3 Der Algorithmus

Zunächst sind des Verständnisses wegen einige wiederkehrende Muster im Programmcode mitsamt einer kurzen Erklärung aufgelistet. Der Code für die Klassen `poly` und `gf_element` befindet sich im Anhang. Der gesamte verwendete Quellcode ist im Übrigen auch in einem Github-Repositorium zu finden.

[10] Laurent 2008, S. 3.

[11] Laurent 2008, S. 3.

[12] Madritsch 2010, S. 5–6.

[13] Weitz 2021, S. 773.

[14] Weitz 2021, S. 68.

[15] Wenn man die Elemente des Körpers umbenennt, zum Beispiel a,b,c,... statt 1,2,3,..., erhält man einen Körper, der zwar funktional vollkommen identisch, aber trotzdem technisch gesehen ein Anderer ist.

[16] Madritsch 2010, S. 6.

[17] Weitz 2021, S. 779.

Abb. 1: Wiederkehrende Muster im Programm

```
# gibt "Hallo, Welt!" in der Konsole aus.
print("Hallo, Welt!")

# definiert eine funktion "code", die als x als Argument entgegennimmt und dessen Nachfolger ausgibt
def code(x):
    # gibt den Wert von x+1 zurück
    return x + 1
# Ein return beendet automatisch die Funktion.
# Die folgenden zwei Funktionen a und b sind funktional identisch.
def a(x):
    if x>=0:
        return 1
    else:
        return 0
def b(x):
    if x>=0:
        return 1
    return 0


# führt code(i) für jede Zahl i im inlkusiven Intervall von 0 bis x - 1 aus
for i in range(x):
    code(i)

# erstellt eine Liste mit allen i im Intervall von 0 bis x-1
liste1 = [i for i in range(x)]

# erstellt eine Liste mit den Werten von code(i) für alle i im Intervall von 0 bis x-1
liste2 = [code(i) for i in range(x)]

# Man kann Listen auch addieren um sie zusammenzufügen
addlists = [5,1,3,2] + [3,3,6] # wird zu [5,1,3,2,3,3,6]

# Fügt die Elemente der Liste zu einem zusammenhängenden String zusammen, wobei zwischen zwei Listenelemente immer das Zeichen zwischen den Anführungszeichen vor dem Punkt kommt (hier: "0-1-Manfred-Fahrrad").
```

```
joined = "-".join( ["0","1","Manfred","Fahrrad"] )

# Wandelt die ganze Zahl x in das Unicode-Zeichen mit dem entsprechenden
    Index um. (z.B. "{0:c}".format(42) → *)
formatted = "{0:c}".format(x)

# Wandelt die ganze Zahl x in eine 8Bit Binärzahl um (z.B. "{0:08b}".
    format(42) → 00101010)
formatted2 = "{0:08b}".format(x)

# Wenn x gleich y
if x == y:
    # mache nichts
    pass
# ansonsten, wenn x ungleich y
elif x != y:
    code(x)
# ansonsten
else:
    pass

# für jedes i in liste1
for i in liste1:
    # falls x kleiner oder gleich 3
    if x <= 3:
        # springe zum nächsten i
        continue
    # ansonsten, wenn x = 0 (mod 5)
    elif x % 5 == 0:
        # beende die Schleife
        break
    code(i)
```

Einige spezifische Muster des Beispielcodes sind hier gezeigt.

```
five = gfe(5) # erstellt ein Element aus GF(2^8) dessen
    Dezimaldarstellung 5 ist

new_Poly = poly( [gfe(4), gfe(2), gfe(0), gfe(1)] ) # erstellt ein
    Polynom über GF(2^8), mit den gegebenen Koeffizienten. Der Koeffizient
```

```
    ganz rechts hat den Exponenten 0.
4
5 # Konstanten, die im Code immer wieder vorkommen:
6 BLKSIZE = 255       # Die Größe eines Nachrichtenblocks
7 ERRCHARS = 16       # Die Anzahl der Byte-Fehler, die maximal korrigiert
    werden können
8 MSGCHARS = BLKSIZE - 2 * ERRCHARS   # Die Anzahl der Bytes, die für die
    Nachricht übrigbleiben
9 ALPHA = gfe(3)      # Eine Primitive Einheitswurzel von GF(2^8)
10 ZERO = poly([gfe(0)])   # Ein Polynom vom Grad 0 mit dem Wert 0
```

3.1 Die Grundidee

Ein Polynom vom Grad n kann durch $n+1$ verschiedene Punkte eindeutig definiert werden.[18] Um bei unserem Beispiel aus der Einleitung, von der CD, die vom Laufwerk eingelesen wird, zu bleiben, orientieren wir uns am CD-Standard. Dieser sieht ein Nachrichtenvolumen von 223 Bytes pro Block und ein Fehlerkorrekturvermögen von 16 Bytes pro Block vor. Dementsprechend benötigen wir für eine Nachricht ein Polynom des Grades 222.

Wenn man nun ein Polynom $m(x)$ vom Grad 222, wie es hier der Fall ist, übertragen will, so braucht man den Wert von $m(x)$ an mindestens 223 Stellen. Will man zudem noch die Möglichkeit haben, bis zu 16 Fehler zu korrigieren, muss man $223+2\cdot 16$ Punkte versenden. Nehmen wir nun an, dass es einen Übertragungsfehler gab und 16 der ursprünglichen Punkte nun falsch sind. Man bekommt also, je nach dem, welche Punkte man wählt, ein anderes Polynom als m. Weil m aus 223 richtigen Punkten bereits eindeutig wiederherstellbar ist, können nicht mehr als 222 richtige Punkte auf dem falschen Polynom liegen. Gibt es bei der Übertragung also maximal 16 Fehler, dann kann man zwar mit 222 richtigen und 16 falschen Punkten ein falsches Polynom bilden, aber mit 223+16 richtigen Punkten, also einem Punkt mehr, immer noch das ursprüngliche Nachrichtenpolynom rekonstruieren.[19] Das heißt, dass die CD sich dank dieses RS-Codes trotz kleinerer Kratzer, welche vielleicht sogar mehrere ganze Bytes hintereinander unkenntlich machen, einwandfrei auslesen lässt.

[18] Weitz 2021, S. 612.

[19] Weitz 2021, S. 770–771.

3.2 Die Vorbereitung

Wir könnten die Werte von Polynomen mit reellen Koeffizienten versenden, das würde aber allein schon an den Rundungsfehlern der Binärdarstellung scheitern. Zudem könnten die Werte des Polynoms – aufgrund der Tatsache, dass man reelle Zahlen auf reelle Zahlen abbildet – technisch gesehen unendlich groß werden, was die benötigte Bandbreite oder den Speicherplatz selbst für einfache Nachrichten unvorhersehbar macht.[20]

Für die Datenübertragung in Form von Bytes, also der für Computer üblichsten Stückelung von Daten, bräuchten wir idealerweise genug Zeichen, sodass wir mit Ihnen Binärzahlen von der Länge 8 Bit repräsentieren können. Das heißt wir bräuchten einen Körper, der uns exakt 2^8 verschiedene Elemente zur Verfügung stellt. Und weil 2 eine Primzahl ist, existiert dieser Körper sogar. Er heißt $GF(2^8)$ und lässt sich wie folgt konstruieren:

Wir benötigen:

1. Ein irreduzibles Polynom $g(x)$ vom Grad n über dem Körper $\mathbb{Z}_p$. Irreduzibel heißt, dass sich das Polynom nicht weiter faktorisieren lässt. Anders gesagt gibt es kein Polynom, durch das man g teilen kann, ohne eine gebrochen rationale Funktion zu erhalten. Dementsprechend bilden irreduzible Polynome ein Analog zu den Primzahlen. In unserem Fall nehmen wir ein Polynom vom Grad 8 über dem Körper $\mathbb{Z}_2$. Dass $g(x) = x^8 + x^4 + x^3 + x + 1$ irreduzibel ist, lässt sich mit folgendem Code einfach nachprüfen[21]:

Abb. 2: Test für ein irreduzibles Polynom

```
1 from sympy import Poly
2 from sympy.abc import x
3 g = Poly(x**8+x**4+x**3+x+1, modulus=2)
4 print(g.is_irreducible)
```

Die Elemente des Körpers wird die Menge aller Polynome vom Grad kleiner als 8 – also aller Reste, die bei der Division mit g entstehen können – sein. Um die Darstellung zu vereinfachen, drücken wir die Polynome durch ihre Koeffizienten aus, welche sich gut als Binärzahlen darstellen lassen.[22]

[20] Weitz 2021, S. 772.

[21] Code adaptiert von Weitz 2021, S. 781. Zu finden im Ordner `Beispielprogramm` unter dem Namen `irreducible.py`

[22] Weitz 2021, S. 781.

2. Eine Methode der Addition: Bei der Addition von zwei Polynomen addiert man die jeweiligen Koeffizienten mit dem selben Exponenten. Da wir in $\mathbb{Z}_2$ rechnen, sieht die Additionstabelle für die Addition zweier Koeffizienten folgendermaßen aus:

+	0	1
0	0	1
1	1	0

Das ist exakt die Wahrheitstabelle für die Boole'sche Operation des exklusiven Oder (XOR).[23] Weil wir diese Operation Koeffizient für Koeffizient anwenden, lässt sich die Addition von zwei Polynomen über $\mathbb{Z}_2$ dementsprechend mit der in jedem Computer eingebauten Bitweisen XOR-Operation ersetzen.

3. Eine Methode der Multiplikation: Hierzu multiplizieren wir zwei Elemente a und b zunächst wie gewöhnliche Polynome über $\mathbb{Z}_2$ und berechnen dann den Rest, der bei einer Division mit g entsteht.

Kürzer geschrieben ist die Multiplikation das Ergebnis von $(a \cdot b) \mod g$.[24]

Das Rechnen in Galoiskörpern unterscheidet sich also kaum vom Rechnen in anderen endlichen Körpern, man muss sich nur daran gewöhnen, dass man mit Polynomen und nicht mit ganzen Zahlen rechnet.

Die Multiplikation lässt sich effizienter implementieren, indem man sich eine primitive Einheitswurzel $\alpha \in GF(2^8)$, zum Beispiel $x+1$ bzw. `11`, nimmt und eine Liste mit allen Potenzen dieser Einheitswurzel erstellt:

k	0	1	2	3	4	5	...
α^k	1	11	101	1111	10001	110011	...

Anschließend entnimmt man aus der Liste die Indizes k_a und k_b, an denen a bzw. b zu finden sind und nimmt dann das Listenelement mit dem Index $(k_a + k_b) \mod 255$.[25]

4. Die Identitäten: Die additive Identität ist `0` und die Multiplikative ist `1`.

5. Die Inversen: Das additive Inverse von $a \in K$ ist a, weil der XOR-Operator bei zwei gleichen Inputs 0 zurück gibt.

[23] Weitz 2021, S. 781.

[24] Weitz 2021, S. 781.

[25] Weitz 2021, S. 782.

Ist $a \in K = \alpha^{k_a}$, dann ist $a^{-1} = \alpha^{k_{a^{-1}}} = \alpha^{255-k_a}$, weil $k_a + k_{a^{-1}} = 0 \pmod{255}$ sein muss.

Man kann Elemente von $GF(2^8)$ unterschiedlich darstellen:

Als Polynom der Form $ax^7 + bx^6 + cx^5 + dx^4 + ex^3 + fx^2 + gx + h$ mit $a,b,c,d,e,f,g,h \in \{0,1\}$,

Als Binärzahl, welche die Koeffizienten darstellt (Ein Koeffizient kann entweder 0 oder 1 sein, deshalb lassen sich Polynome vom Grad 7 besonders gut als 8-Bit-Binärzahl darstellen.)

Als Dezimalzahl, indem man die Binärdarstellung als Zahl behandelt und ins Dezimalsystem umrechnet. Da dies die am wenigsten platz-intensive Methode ist, wird das die im Folgenden am meisten benutzte Darstellung sein.

3.3 Die Verschlüsselung

Nehmen wir eine Nachricht, zum Beispiel „hallo“. Diese Zeichenkette konvertieren wir zuerst in 8-Bit lange Binärzahlen, um sie mit $GF(2^8)$ kompatibel zu machen. Anschließend setzen wir diese Elemente als Koeffizienten in unser Nachrichtenpolynom $m(x)$ ein und erweitern dieses gegebenenfalls noch mit genügend vielen Nullkoeffizienten, um auf den Grad 222 (für 223 Bytes an Information) zu kommen. Das Polynom sieht dann folgendermaßen aus:

$$m(x) = 104x^{222} + 97x^{221} + 108x^{220} + 108x^{219} + 111x^{218} + 0x^{217} + \cdots + 0x^0$$

Diese Aufgabe übernimmt folgender Programmcode für uns:

Abb. 3: Erstellen des Nachrichtenpolynoms

```
1 msg = poly([gfe(ord(i)) for i in message]           # Die Nachricht
2     + [gfe(0)] * (MSGCHARS - len(message)))         # Das "Füllmaterial"
          um auf die richtige Länge zu kommen
```

Dieses eben erstellte Polynom $m(x)$ werten wir zunächst an 255 verschiedenen, vorher beidseitig gleichermaßen festgelegten Stellen aus. Wir verwenden dazu die Potenzen von α, also alle $x \in \{\alpha^k | 0 \leq k \leq 254\}$.[26] Welche Reihenfolge wir dann für diese Werte nehmen, ist egal.

[26] Weitz 2021, S. 773.

Wichtig ist nur, dass wir erstens die 0 nicht als Stelle nehmen – weil sonst die später wichtige Gleichung $x^{255} = 1$ für diese eine Stelle nicht stimmt – und dass zweitens beide Seiten die selbe Reihenfolge verwenden, damit die x-Werte bei der Interpolation zu den y-Werten passen. Die Werte von $m(\alpha^k)$ sehen folgendermaßen aus (k beginnt oben links bei 0 wird erst von links nach rechts und dann von oben nach unten größer):

102	196	243	245	245	98	179	29	165	186	242
204	223	91	19	128	136	39	195	64	96	26
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
207	217	252	75	4	33	134	67	183	123	255
102	207									

Tabelle 2: Die Werte von $m(x)$

Der Programmcode dazu sieht so aus:

Abb. 4: Auswerten des Nachrichtenpolynoms

```
def encode(message):
    # global lässt uns auf globale Variablen zugreifen
    global MSGCHARS, ERRCHARS, BLKSIZE, ALPHA
    # hier wird das Nachrichtenpolynom erstellt
    msg = poly([gfe(ord(i)) for i in message]
        + [gfe(0)] * (MSGCHARS - len(message)))
    # gibt eine Liste mit den Werten von msg an den Stellen α^k zurück
    return [msg(ALPHA ** (i)) for i in range(BLKSIZE)]
```

Diese Liste schicken wir weiter an den Empfänger. Mit einem Zufallsgenerator verteilen wir schließlich Fehler in der Nachricht:

Abb. 5: Der Fehlergenerator

```
def inserrRB(message, x):
    msg = [i for i in message]
    errcache = []
    for i in range(x):
        z = ran.randint(0,254)
        # Garantiert eine einzigartige Zufallszahl
        while z in errcache:
            z = ran.randint(0,254)
```

```
        msg[z] += gfe(ran.randint(1,254))
    debug(1,"Errors inserted at positions:",*(i for i in errcache))
    return msg
```

3.4 Die Entschlüsselung

3.4.1 Schritt 1: Die erste Interpolation

Aus der erhaltenen Liste versuchen wir im ersten Schritt, ein Polynom wiederherzustellen.[27] Das gelingt mit der Lagrange-Interpolation. Eine Beispielimplementierung ist wie folgt:

Abb. 6: Beispielcode für eine Lagrange-Interpolation

```
def lagrange(values, indices):
    global ALPHA, ZERO
    # Erstellt die Liste mit den passenden Stellen
    x_vals = [ALPHA**i for i in indices]
    # Initialisiert ein Polynom L(x) = 0, welches später das
       Gesamtergebnis wird
    L = ZERO
    for i, x_i in enumerate(x_vals):
        # initialisiert den Zähler von l_i als f(x) = 1
        zaehler = x(0)
        # initialisiert den Nenner von l_i als eine leere Liste
        nenner = []
        for j, x_j in enumerate(x_vals):
            if j != i:
                # Multipliziert den Zähler in jedem Schritt mit (x-x_j)
                zaehler = zaehler.flshift(1) + zaehler.scale(x_j)
                # Fügt (x_i-x_j) zum Nenner hinzu
                nenner.append((x_i-x_j))
            else:
                continue
        # multipliziert alle Elemente der Nennerliste auf einen Schlag
        nenner = gfe.prod(nenner)
        l_i = zaehler.scale(values[i]*nenner.inv())
        # Addiert l_i zum Gesamtergebnis
        L+=l_i
    return L
```

[27] Weitz 2021, S. 773–774.

Wir haben zwar nur die y-Werte des Polynoms bekommen, aber die x-Gegenstücke können wir uns mit α^k leicht selbst generieren (siehe Code).
Angenommen, die ersten 16 stellen unserer Nachricht wurden verändert und lauten nun:

62 186 239 232 219 249 15 250 84 59 250 236 91 176 81 164

Wenn wir an dieser veränderten Liste an Werten die Lagrange-Interpolation durchführen, erhalten wir folgendes Polynom:

$$\begin{aligned} r(x) = {} & 184x^{254} + 7x^{253} + 184x^{252} + 39x^{251} + 128x^{250} + 204x^{249} + 247x^{248} \\ & + 52x^{247} + 227x^{246} + 89x^{245} + 102x^{244} + 102x^{243} + 39x^{242} + 248x^{241} \\ & + 153x^{240} + 73x^{239} + \cdots + 255x^2 + 41x^1 + 152x^0 \end{aligned}$$

Es ist offensichtlich, dass r nicht unser Nachrichtenpolynom ist, weil sein Grad anders ist als der, den wir von m erwarten. Wäre r unser gesuchtes Polynom, dann hätte es den Grad 222. Der Grad von r muss außerdem größer sein als der von m, wenn die Nachricht fehlerbehaftet ist: Nehmen wir an, dass es nicht mehr als 16 Fehler bei der Übertragung gibt, so finden sich in unserer Nachricht 223 Punkte, die auf jeden Fall auf m liegen. [28] Da m das einzige Polynom vom Grad ≤ 222 ist, das durch diese 223 Punkte geht[29], muss das Polynom, das durch diese Punkte **und** beliebige zusätzliche, nicht auf m liegenden Punkte geht, einen Grad von mindestens 223 haben. Deshalb wissen wir zweifelsfrei, dass r nicht m ist, wenn der Grad von r größer als 222 ist.

3.4.2 Schritt 2: Die Fehlerfindung

Das sogenannte Fehlerpolynom $e(x)$ ist definiert als $r(x) - m(x)$[30] und hat somit den selben Grad, wie r. Weil $r = m + e$ ist, hat e an allen Stellen, an denen $r = m$ ist, eine Nullstelle, weil sich m dort nach Addition mit e nicht verändert hat. Dementsprechend lässt es sich als Produkt von Linearfaktoren $x - x_i$ beschreiben, wobei x_i die Stellen sind, an denen es keinen Fehler gab.
Nehmen wir uns ein neues Polynom $n = x^{255} - 1$, welches für alle $x \in GF(2^8)\backslash\{0\}$ gleich 0 ist. Das ist der Fall, weil x^{255} für alle $x \neq 0$ gleich 1 ist[31], was sich mit folgendem Code

[28] Es gibt noch 16 weitere solche Werte, aber da m bereits durch die 223 richtigen Werte eindeutig beschrieben ist, ignorieren wir sie vorerst einmal.

[29] Weitz 2021, S. 612.

[30] Weitz 2021, S. 774.

[31] Weitz 2021, S. 775.

einfach nachweisen lässt:

Abb. 7: $n(x)$=0 $\forall x \in GF(2^8)\backslash\{0\}$

```
# set() nimmt eine Liste und fasst alle mehrfach vorkommenden Elemente zu
    einem zusammen
# Wenn also set(x) die länge 1 hat, dann sind alle Elemente in x gleich
print( len( set( [gfe(i)**255 for i in range(1,256)] ) ) == 1 )
```

e und n haben die Nullstellen gemeinsam, an denen kein Fehler aufgetreten ist. Das kleinste gemeinsame Vielfache (kgV) von e und n hat, weil es ein Vielfaches von n ist, die selben Nullstellen, wie n. Das heißt, für $d(x)$ mit $de = wm$ sind die Nullstellen die Stellen, an denen die Nachricht Fehler hat.[32] Glücklicherweise gibt es einen Algorithmus, um genau dieses d zu berechnen. Der folgende Code ist eine auf das Nötigste herunter gebrochene Version einer veränderten Form des erweiterten Euklidischen Algorithmus, welche neben dem kgV unter Anderem auch den ggT berechnet.[33][34]

Abb. 8: kgV Berechnen

```
def kgV(a, b):
    if a.deg < b.deg:
        a, b = b, a
    d0, d = ZERO, x(0)
    while a.deg > MSGCHARS+ERRCHARS:
        tmp = a.divmod(b)
        f = tmp[0]
        a, b = b, tmp[1]
        d0, d = d, f*d+d0
    return d
```

Der Rückgabewert dieser Funktion ist unser oben gesuchtes d. In unserem Fall sieht d so aus:

$$d(x) = 248x^{16} + 109x^{15} + 146x^{14} + 47x^{13} + 147x^{12} + 125x^{11} + 137x^{10} + 152x^9 + 95x^8 + 11x^7 + 176x^6 + 159x^5 + 101x^4 + 99x^3 + 202x^2 + 13x^1 + 109$$

Nun gilt es, die Nullstellen dieses Polynoms zu ermitteln. Dazu iterieren wir einfach über

[32] Weitz 2021, S. 776.
[33] Weitz 2021, S. 776.
[34] Weitz 2024, S. 24.

alle Stellen, an denen d definiert ist, setzen die x-Werte in d ein und notieren sie uns, wenn $d(x) = 0$ ist. Die Funktion, die die Fehlerstellen findet, sieht dann so aus:

Abb. 9: Fehlerstellen finden

```
def errloc(res_poly):
    global BLKSIZE, ALPHA
    n = x(BLKSIZE) - x(0)
    Lambda = kgV(n, res_poly)
    X = []
    for i in range(BLKSIZE):
        if Lambda(ALPHA**(i)) == 0:
            X.append(i)
    return X
```

Für unser d gibt die Funktion richtigerweise das Intervall $[0;15]$ zurück. Nachdem wir nun die Fehlerstellen identifiziert haben, können wir diese Stellen der fehlerbehafteten Werteliste bei der Interpolation einfach überspringen.

3.4.3 Schritt 3: Die zweite Interpolation

Der Code zum Auswählen der Stellen für die zweite Interpolation nimmt nicht alle verfügbaren richtigen Stellen, sondern nur so viele, wie für die Wiederherstellung von m unbedingt notwendig sind. Das spart etwas Rechenzeit im Vergleich zur Interpolation mit allen möglichen Stellen.

Abb. 10: Erneutes Interpolieren mit weniger Werten

```
    err_indices = [i for i in errloc(respoly)]
    indices = []
    # Diese Schleife füllt eine Liste mit exakt so vielen richtigen
       Indizes auf, wie man benötigt um m(x) wiederherzustellen
    i = 0
    while len(indices) < MSGCHARS:
        if i not in err_indices:
            indices.append(i)
        i += 1
    res = lagrange([message[i] for i in indices], indices)
```

Die vollständige Dekodierfunktion ist nicht viel länger:

```
def decode(message):
    global BLKSIZE, ERRCHARS, ALPHA, errcache
    respoly = lagrange(message, range(BLKSIZE))
    err_indices = []
    if len(set(respoly.coeffs[(MSGCHARS+1):])) == 1 and respoly.coeffs
        [-1] == 0:
        return "".join(["{0:c}".format(i.val) for i in respoly.coeffs]).
            strip("\00")
    else:
        err_indices = [i for i in errloc(respoly)]
    indices = []
    i = 0
    while len(indices) < MSGCHARS:
        if i not in err_indices:
            indices.append(i)
        i += 1
    res = lagrange([message[i] for i in indices], indices)
    ret = "".join(["{0:c}".format(i.val) for i in res.coeffs]).strip("\00
        ")
    return ret
```

3.4.4 Grenzen unseres Codes

Die meisten CDs sind aber größer als 223 Bytes. Wollen wir mit unserem Code Nachrichten versenden, die mehr als 223 Zeichen lang sind, müssen wir die Nachricht in verschiedene Blöcke Aufteilen und diese einzelnen Blöcke separat kodieren. Eine Beispiellösung wäre Folgendes:

Man erstellt eine neue `encode()`-Funktion und benennt die Alte in `encode_block()` um. Die neue `encode()` Funktion teilt die Nachricht in 223-Zeichen-Blöcke auf und kodiert sie der Reihe nach.

Abb. 11: Die neue encode()-Funktion

```
def encode(message):
    blocks = []
    while len(message)>223:
        blocks.append(message[:223])
        message = message[223:]
    blocks.append(message)
```

```
7       enc_blocks = []
8       for i, block in enumerate(blocks):
9           enc_blocks.append(encode_block(block))
10      return enc_blocks
```

Das Dekodieren funktioniert ähnlich:

Abb. 12: Die neue decode()-Funktion

```
1  def decode(blocks):
2      dec_blocks = []
3      for i, block in enumerate(blocks):
4          debug(1, "Decoding block Nr.", i+1)
5          dec_blocks.append(decode_block(block))
6      return "".join(dec_blocks)
```

Sie gibt allerdings keine Liste, sondern die rekonstruierte Zeichenkette wieder aus. Mit Hilfe dieser Zerstückelung könnte man beispielsweise sogar ein ganzes Filmskript kodieren.

4 Zusammenfassung

Reed-Solomon-Codes bieten eine gute Möglichkeit, fehlerresistente digitale Nachrichten mit wenig Redundanz zu übertragen. Sie sind viel komplexer als andere, offensichtlichere Fehlerkorrekturmethoden, sind aber durch ihre hohe Informationsdichte deutlich effizienter und dadurch langfristig günstiger. Ihnen ist zum Beispiel zu verdanken, dass man Musikalben nicht auf mehrere CDs verteilen muss, weil sonst der Speicherplatz nicht ausreicht.

Die Implementierung insbesondere des Dekodierverfahrens ist sehr komplex und erfordert umfangreiche mathematische Vorkenntnisse und – je nach Art der Implementierung – Vorwissen über Programmierung bzw. Schaltungslogik. Ist ein RS-Code jedoch einmal implementiert, so kann man ihn mit nur wenig Veränderungen für andere Datentypen verwenden. Wir haben ASCII-Zeichen übertragen, man kann mit den selben Bits jedoch auch bereits kompilierte Programme, Audiodateien oder Bilder übertragen und muss nur die Interpretation der Daten anpassen.

A Restlicher Programmcode

Diese Code-Ausschnitte wurden nicht explizit behandelt, sie sind aber trotzdem hier angefügt, weil die Art, wie bestimmte Funktionen implementiert sind ein Stück weit bestimmt, wie die oben behandelten Funktionen aussehen. Jemandem ohne Zugriff auf die ursprünglichen Programmdateien soll dies eine Möglichkeit bieten, die Implementierung besser nachvollziehen zu können.

A.1 Helferfunktionen

Abb. 13: debug() & Co.

```
1  loglevel = 0
2
3  def set_loglevel(x):
4      global loglevel
5      loglevel = x
6
7  # Erbt es, die Menge der vom Programm mitgeteilten Information zu steuern
8  def debug(x, *args, **kwargs):
9      global loglevel
10     if loglevel>=x:
11         print(*args, *kwargs)
```

Abb. 14: x()

```
1  # Erleichtert es, Monome vom Grad x zu erstellen
2  def x(x):
3      return poly([gf_element(1)]+[gf_element(0)]*x)
```

A.2 Die Klassen „gf_element“ und „poly“

Abb. 15: Die gf_element-Klasse

```
1  class gf_element():
2      # Diese Listen wurden übernommen von: https://github.com/brownan/Reed
           -Solomon/blob/master/ff.py
3      # Die Idee, Eigene Klassen für die Galoiskörperelemente bzw. Polynome
           zu erstellen und die eingebauten Operatoren zu überschreiben
           stammt auch von dort, der genaue Inhalt der Funktionen wurde
           allerdings eigenständig erarbeitet.
```

```
    m_lut = (1, 3, 5, 15, 17, 51, 85, 255, 26, 46, 114, 150, 161, 248,
        19, 53, 95, 225, 56, 72, 216, 115, 149, 164, 247, 2, 6, 10, 30,
        34, 102, 170, 229, 52, 92, 228, 55, 89, 235, 38, 106, 190, 217,
        112, 144, 171, 230, 49, 83, 245, 4, 12, 20, 60, 68, 204, 79, 209,
        104, 184, 211, 110, 178, 205, 76, 212, 103, 169, 224, 59, 77, 215,
         98, 166, 241, 8, 24, 40, 120, 136, 131, 158, 185, 208, 107, 189,
        220, 127, 129, 152, 179, 206, 73, 219, 118, 154, 181, 196, 87,
        249, 16, 48, 80, 240, 11, 29, 39, 105, 187, 214, 97, 163, 254, 25,
         43, 125, 135, 146, 173, 236, 47, 113, 147, 174, 233, 32, 96, 160,
         251, 22, 58, 78, 210, 109, 183, 194, 93, 231, 50, 86, 250, 21,
        63, 65, 195, 94, 226, 61, 71, 201, 64, 192, 91, 237, 44, 116, 156,
         191, 218, 117, 159, 186, 213, 100, 172, 239, 42, 126, 130, 157,
        188, 223, 122, 142, 137, 128, 155, 182, 193, 88, 232, 35, 101,
        175, 234, 37, 111, 177, 200, 67, 197, 84, 252, 31, 33, 99, 165,
        244, 7, 9, 27, 45, 119, 153, 176, 203, 70, 202, 69, 207, 74, 222,
        121, 139, 134, 145, 168, 227, 62, 66, 198, 81, 243, 14, 18, 54,
        90, 238, 41, 123, 141, 140, 143, 138, 133, 148, 167, 242, 13, 23,
        57, 75, 221, 124, 132, 151, 162, 253, 28, 36, 108, 180, 199, 82,
        246, 1)
    index_lut = (None, 0, 25, 1, 50, 2, 26, 198, 75, 199, 27, 104, 51,
        238, 223, 3, 100, 4, 224, 14, 52, 141, 129, 239, 76, 113, 8, 200,
        248, 105, 28, 193, 125, 194, 29, 181, 249, 185, 39, 106, 77, 228,
        166, 114, 154, 201, 9, 120, 101, 47, 138, 5, 33, 15, 225, 36, 18,
        240, 130, 69, 53, 147, 218, 142, 150, 143, 219, 189, 54, 208, 206,
         148, 19, 92, 210, 241, 64, 70, 131, 56, 102, 221, 253, 48, 191,
        6, 139, 98, 179, 37, 226, 152, 34, 136, 145, 16, 126, 110, 72,
        195, 163, 182, 30, 66, 58, 107, 40, 84, 250, 133, 61, 186, 43,
        121, 10, 21, 155, 159, 94, 202, 78, 212, 172, 229, 243, 115, 167,
        87, 175, 88, 168, 80, 244, 234, 214, 116, 79, 174, 233, 213, 231,
        230, 173, 232, 44, 215, 117, 122, 235, 22, 11, 245, 89, 203, 95,
        176, 156, 169, 81, 160, 127, 12, 246, 111, 23, 196, 73, 236, 216,
        67, 31, 45, 164, 118, 123, 183, 204, 187, 62, 90, 251, 96, 177,
        134, 59, 82, 161, 108, 170, 85, 41, 157, 151, 178, 135, 144, 97,
        190, 220, 252, 188, 149, 207, 205, 55, 63, 91, 209, 83, 57, 132,
        60, 65, 162, 109, 71, 20, 42, 158, 93, 86, 242, 211, 171, 68, 17,
        146, 217, 35, 32, 46, 137, 180, 124, 184, 38, 119, 153, 227, 165,
        103, 74, 237, 222, 197, 49, 254, 24, 13, 99, 140, 128, 192, 247,
        112, 7)
    def __init__(self, x):
        if x<0:
```

```
8               x*=-1
9           self.val = x%256
10          self.char = "{0:c}".format(x)
11
12      def __add__(a, b):
13          if b == 0:
14              return a
15          return gf_element(a.val^b.val)
16
17      def __sub__(a,b):
18          return a.__add__(b)
19
20      def __radd__(a,b):
21          return a.__add__(b)
22
23      def __mul__(a, b):
24          if a == 0 or b == 0:
25              return gf_element(0)
26          try:
27              return gf_element(gf_element.m_lut[(gf_element.index_lut[a.
                    val]+gf_element.index_lut[b.val])%255])
28          except IndexError:
29              print(a.val)
30              exit(1)
31
32      def __pow__(a, x):
33          if a == 0:
34              return gf_element(0)
35          return gf_element(gf_element.m_lut[(gf_element.index_lut[a.val]*x
                )%255])
36
37      def inv(self):
38          if self.val == 0:
39              return gf_element(0)
40          return gf_element(gf_element.m_lut[255-gf_element.index_lut[self.
                val]])
41
42
43      def __truediv__(a, b):
44          if a == 0:
```

```
45              return gf_element(0)
46          if b == 0:
47              raise ValueError("Cannot divide by 0")
48          return a*b.inv()
49
50      def __eq__(self, other):
51          if other is None:
52              return False
53          if isinstance(other, int):
54              return self.val == other
55          return self.val == other.val
56
57      def __repr__(self):
58          return self.__str__()
59
60      def __str__(self):
61          return str(self.val)
62
63      def __hash__(self):
64          return hash(self.val)
```

Abb. 16: Die poly-Klasse

```
1  # Alle Funktionen der Form __name__ überschreiben Python-interne
       Funktionen, sodass sich klassenspezifische Methoden zum Addieren,
       Multiplizieren, etc. definieren lassen.
2  class poly():
3      def __init__(self, coefficients):
4          if coefficients == 0:
5              self.coeffs = [gf_element(0)]
6          else:
7              self.coeffs = [i for i in coefficients]
8          self.udeg()
9
10     def __add__(a, b):
11         ac = a.coeffs
12         bc = b.coeffs
13         match a.deg>b.deg:
14             case True:
15                 bc = [gf_element(0)]*(a.deg-b.deg)+bc
16             case False:
```

```
17                  ac = [gf_element(0)]*(b.deg-a.deg)+ac
18          return poly(x+y for x,y in zip(ac,bc))
19
20      __sub__ = __add__
21
22      def __mul__(a, b):
23          if isinstance(b, gf_element):
24              return poly(i*b for i in a.coeffs)
25          ac = a.coeffs
26          bc = b.coeffs
27          res = [gf_element(0) for _ in range(a.deg+b.deg+1)]
28          for i, ae in enumerate(ac):
29              for j, be in enumerate(bc):
30                  res[i+j] += ae*be
31          return poly(res)
32
33      def divmod(a, b):
34          dividend = a.coeffs.copy()
35          divisor = b.coeffs.copy()
36          quot = []
37          while len(dividend) >= len(divisor):
38              if dividend[0] == 0:
39                  quot.append(dividend.pop(0))
40                  continue
41              quot.append(dividend[0]/divisor[0])
42              for i in range(len(divisor)):
43                  dividend[i] -= divisor[i]*quot[-1]
44              dividend.pop(0)
45          return poly(quot), poly(dividend)
46
47      def __truediv__(self, other):
48          if isinstance(other, gf_element):
49              return poly([i/other for i in self.coeffs])
50          return self.divmod(other)[0]
51
52      def __mod__(self, other):
53          return self.divmod(other)[1]
54
55      def flshift(self, x):
56          c = self.coeffs.copy()
```

```
57          c += [gf_element(0)]*x
58          return poly(c)
59
60      def udeg(self):
61          self.deg = len(self.coeffs)-1
62
63      def eval(self, x):
64          return sum([a*x**(self.deg-i) for i, a in enumerate(self.coeffs)
                ])
65
66      def __call__(self, x):
67          return self.eval(x)
68
69      def __eq__(self, other):
70          if isinstance(other, int):
71              return self.coeffs[0] == other and self.coeffs[-1] == other
72          return self.coeffs == other.coeffs
73
74      def __str__(self):
75          res = (f"{x}x^{self.deg-y}" for y, x in enumerate(self.coeffs))
76          return "+".join(res)
77
78      def __repr__(self):
79          return str(self.coeffs)
80
81      def __neg__(self):
82          return poly(-i for i in self.coeffs)
83
84      def scale(self, x):
85          self.coeffs = [i*x for i in self.coeffs]
86          return self
```

A.3 Beispielcode für die Kodierung

Abb. 17: Das Beispielprogramm

```
1 from gf256 import gf_element as gfe, poly, debug, set_loglevel, x
2 import time
3 import random as ran
4
```

```
BLKSIZE = 255
ERRCHARS = 16
MSGCHARS = BLKSIZE - 2 * ERRCHARS
ALPHA = gfe(3)
ZERO = poly([gfe(0)])

def encode_block(message):
    global MSGCHARS, ERRCHARS, BLKSIZE, ALPHA
    debug(2, "Constructing message polynomial...")
    msg = poly([gfe(ord(i)) for i in message] + [gfe(0)] * (MSGCHARS -
        len(message)))
    debug(3, msg.coeffs)
    debug(2, "Encoding message...")
    return [msg(ALPHA ** (i)) for i in range(BLKSIZE)]

def encode(message):
    debug(1,"Organizing the message into blocks of 223...")
    blocks = []
    while len(message)>223:
        blocks.append(message[:223])
        message = message[223:]
    blocks.append(message)
    enc_blocks = []
    for i, block in enumerate(blocks):
        debug(1, "Encoding block Nr.", i+1)
        enc_blocks.append(encode_block(block))
        debug(1, "done")
    return enc_blocks

def decode_block(message):
    global BLKSIZE, ERRCHARS, ALPHA, errcache
    debug(1, "Receiving block")
    respoly = lagrange(message, range(BLKSIZE))
    err_indices = []
    if respoly.deg == 222:
        debug(1, "No errors detected.")
        return "".join(["{0:c}".format(i.val) for i in respoly.coeffs]).
            strip("\00")
    else:
        debug(1, "The message has errors")
```

```
43          err_indices = [i for i in errloc(respoly)]
44      indices = []
45      i = 0
46      while len(indices) < MSGCHARS:
47          if i not in err_indices:
48              indices.append(i)
49          i += 1
50      res = lagrange([message[i] for i in indices], indices)
51      ret = "".join(["{0:c}".format(i.val) for i in res.coeffs]).strip("\00
            ")
52      return ret
53
54  def decode(blocks):
55      debug(1, "Decoding the Message.")
56      dec_blocks = []
57      for i, block in enumerate(blocks):
58          debug(1, "Decoding block Nr.", i+1)
59          dec_blocks.append(decode_block(block))
60      return "".join(dec_blocks)
61
62
63  def lagrange(values, indices):
64      global ALPHA, ZERO
65      debug(1, "Interpolating the message polynomial...")
66      x_vals = [ALPHA**i for i in indices]
67      L = ZERO
68      for i, x_i in enumerate(x_vals):
69          if i%30 == 0:
70              debug(1, "{0:.2f}%".format(100*i/len(x_vals)))
71          zaehler = x(0)
72          nenner = []
73          for j, x_j in enumerate(x_vals):
74              if j != i:
75                  zaehler = zaehler.flshift(1) + zaehler.scale(x_j)
76                  nenner.append((x_i-x_j))
77              else:
78                  continue
79          nenner = gfe.prod(nenner)
80          l_i = zaehler.scale(values[i]*nenner.inv())
81          L+=l_i
```

```
82      debug(1, "100.00%")
83      return L
84
85  def kgV(a, b):
86      if a.deg < b.deg:
87          a, b = b, a
88      a0 = a
89      d0, d = ZERO, x(0)
90      while a.deg > MSGCHARS+ERRCHARS:
91          tmp = a.divmod(b)
92          f = tmp[0]
93          a, b = b, tmp[1]
94          d0, d = d, f*d+d0
95      return d
96
97  def errloc(res_poly):
98      global BLKSIZE, ALPHA
99      debug(1, "Locating errors...")
100     n = x(BLKSIZE) - x(0)
101     Lambda = kgV(n, res_poly)
102     debug(2, "deg(Lambda) =",Lambda.deg)
103     X = []
104     for i in range(BLKSIZE):
105         if Lambda(ALPHA**(i)) == 0:
106             X.append(i)
107     debug(1, "Errors found at bytes: ", [i for i in X])
108     return X
109
110 def inserrRB(message, x):
111     msg = [i for i in message]
112     errcache = []
113     for i in range(x):
114         z = ran.randint(0,254)
115         while z in errcache:
116             z = ran.randint(0,254)
117         msg[z] += gfe(ran.randint(1,254))
118          #msg[i] += (gfe(i+1)/(gfe(i+24)))
119     debug(1,"Errors inserted at positions:",*(i for i in errcache))
120     return msg
121
```

```
122 def inserrRS(blocks, x):
123     errcache = []
124     for i in range(x):
125         z = ran.randint(0,254)
126         while z in errcache:
127             z = ran.randint(254)
128         errcache.append(z)
129     for block in blocks:
130         for i in errcache:
131             block[i] += gfe(ran.randint(1,254))
132 
133 set_loglevel(1)
134 print("Welcome. Please enter your message:")
135 message = input()
136 enc_blocks = encode(message)
137 print("Would you like to insert some errors into the blocks?")
138 response = input("[y/N]")
139 if not "y" in response.lower():
140     print(decode(enc_blocks))
141     exit(0)
142 print("Would you like to insert random errors at:\n \
143     \trandom bytes for every block (rb)?\n\
144     \tthe same set of up to 16 random bytes applied to every block (rs)?"
        )
145 response = input().lower()
146 if response == "rb":
147     errnums = ran.randint(1,16)
148     print("Inserting errors at",errnums,"random positions in each block")
149     for block in enc_blocks:
150         block = inserrRB(block, errnums)
151 elif response == "rs":
152     errnums = ran.randint(1,16)
153     print("Inserting errors at the same",errnums,"positions in every
        block.")
154     enc_blocks = inserrRS(enc_blocks, errnums)
155 
156 print("Errors inserted")
157 print(decode(enc_blocks))
```

Abbildungsverzeichnis

1 Wiederkehrende Muster im Programm 6
2 Test für ein irreduzibles Polynom 9
3 Erstellen des Nachrichtenpolynoms 11
4 Auswerten des Nachrichtenpolynoms 12
5 Der Fehlergenerator 12
6 Beispielcode für eine Lagrange-Interpolation 13
7 $n(x)$=0 $\forall x \in GF(2^8)\backslash\{0\}$ 15
8 kgV Berechnen 15
9 Fehlerstellen finden 16
10 Erneutes Interpolieren mit weniger Werten 16
11 Die neue encode()-Funktion 17
12 Die neue decode()-Funktion 18
13 debug() & Co. 19
14 x() 19
15 Die gf_element-Klasse 19
16 Die poly-Klasse 22
17 Das Beispielprogramm 24

Tabellenverzeichnis

2 Die Werte von $m(x)$ 12

Literatur

[1] Daryl Deford. „Lagrange Interpolation“. Supplementary Notes for the Course Math 8: Calculus of Functions of one and Several Variables. 25. Sep. 2015. URL: https://people.csail.mit.edu/ddeford/Lagrange_Interpolation.pdf (besucht am 21. 10. 2025).

[2] Anne Kværnø. „Polynomial interpolation: Lagrange interpolation“. Lecture notes for TMA4125/4130/4135 Mathematics 4N/D. 14. Jan. 2021. URL: https://www.math.ntnu.no/emner/TMA4130/2021h/lectures/LagrangeInterpolation.pdf (besucht am 05. 02. 2025).

[3] Hoeltgen Laurent. „Konstruktion und Struktur endlicher Körper“. Bachelorarbeit. Université du Luxembourg, 28. Mai 2008. URL: https://www.mia.uni-saarland.de/hoeltgen/FiniteFields.pdf (besucht am 02. 02. 2025).

[4] Manfred Madritsch. „Endliche Körper und Codierung“. Mitschrift. Institut für Mathematik A Technische Universität Graz, 2010. URL: https://www.math.tugraz.at/~wagner/FFC.pdf (besucht am 02. 02. 2025).

[5] Edmund Weitz. *Konkrete Mathematik (nicht nur) für Informatiker: Mit vielen Grafiken und Algorithmen in Python*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2021. ISBN: 978-3-662-62617-7 978-3-662-62618-4. DOI: 10.1007/978-3-662-62618-4. URL: http://link.springer.com/10.1007/978-3-662-62618-4.

[6] Edmund Weitz. *Lösungen zu ausgewählten Aufgaben aus Weitz: "Konkrete Mathematik (nicht nur) für Informatiker"*. 2024. URL: https://weitz.de/KMFI/9783662626177_Loesungen.pdf (besucht am 21. 05. 2025).

Eigenständigkeitserklärung

Ich erkläre hiermit, dass ich die Seminararbeit ohne fremde Hilfe angefertigt und nur die im Literaturverzeichnis angeführten Quellen und Hilfsmittel benützt habe.

Ort, Datum Unterschrift des Verfassers