Kurs 41.4934
Codierungstheorie
Praktikumsbericht

Eliah Vogel, 761013
Ramon Walther, 761312
Niklas A.J. Werner, 762056

10. September 2022

Hinweis:
Nachfolgend wird unter anderem die Implementierung des Praktikums in Form von Python-Code abgedruckt. Dieser Code enthält allerdings nur die, für die jeweilige Aufgabe, relevanten Bestandteile. Zusätzlich existieren zu Jeder, der nachfolgend bearbeiteten Aufgaben, vordefinierte Testfälle. Diese wurden in Form von automatisierten Unit-Tests in das Projekt integriert. Diese Tests, sowie der gesamte Quellcode des Praktikums, können im dazugehörigen Repository des hochschulinternen GitLab eingesehen werden. Zu finden unter dem folgenden Link:
`https://code.fbi.h-da.de/ct-braun-gruppe-vww/praktikum/`
Sollten Probleme beim Zugriff bestehen, informieren Sie bitte die Projektmitglieder.

Lösung 1: Multiplikationstabelle

Pseudocode

1. Eingabe: Zahl `e` für die gilt, $2 \leq e \leq 8$
2. Passendes irreduziebles Polynom wählen
3. Multiplikationstabelle als leere Tabelle initialisieren
4. Jede Stelle der Tabelle, mittels Multiplikation der Indizes als Polynome, berechnen

Programmcode und Dokumentation

Programmcode

```
ips = [  # Irreducible Polynomial Lookup Table
    '',
    '',
    111,  # 2
    1101,  # 3
    11001,  # 4
    100101,  # 5
    1100001,  # 6
    11000001,  # 7
    100011101,  # 8
    1000010001,  # 9
    10000001001,  # 10
]


class P:  # Polynomial
    def __init__(self, value):
        self.value = str(value)

    def __repr__(self):
        return self.value

    def __len__(self):
        return len(self.value)

    def __eq__(p1, p2):
        p1 = p1.value.lstrip('0')  # Remove leading zeros
        p2 = p2.value.lstrip('0')
        return p1 == p2

    def __add__(p1, p2):
        width = max(len(p1.value), len(p2.value))
        p1, p2 = p1.pad(width), p2.pad(width)
        result = ''
        for i in range(width):
```

```
            pv1, pv2 = int(p1[i]), int(p2[i])
            result += str(pv1 + pv2)
        return P(result)

    def __sub__(p1, p2):
        width = max(len(p1.value), len(p2.value))
        p1, p2 = p1.pad(width), p2.pad(width)
        result = ''
        for i in range(width):
            pv1, pv2 = int(p1[i]), int(p2[i])
            result += str(pv1 - pv2)
        result = result.lstrip('0')  # Remove leading zeros
        return P(result)

    def __mul__(p1, p2):
        pl1, pl2 = len(p1.value), len(p2.value)
        width = pl1 + pl2 - 1
        result = [0] * width
        for i in range(pl1):
            for j in range(pl2):
                pv1, pv2 = int(p1.value[i]), int(p2.value[j])
                result[i + j] += pv1 * pv2
        result = ''.join(str(x) for x in result)  # List to string
        result = result.lstrip('0')  # Remove leading zeros
        return P(result or '0')

    def __truediv__(p1, p2):
        diff = len(p1.value) - len(p2.value)
        if diff < 0:
            return [P('0'), p1]

        pv1, pv2 = int(p1.value[0]), int(p2.value[0])
        factor = int(pv1 / pv2)
        pr = P(str(factor) + '0' * diff)
        pmul = pr * p2
        psub = (p1 - pmul).abs()

        if psub == P('0'):
            return [pr, P('0')]

        pdiv = psub / p2
        return [pr + pdiv[0], pdiv[1]]

    def __floordiv__(p1, p2):
        return (p1 / p2)[0]

    def __mod__(p1, p2):
        mod = (p1 / p2)[1]
        return mod

    def mul(p, factor: int):
        result = ''
        for i in range(len(p.value)):
            pv = int(p.value[i])
            result += str(int(pv * factor))
        return P(result)


```

```
    def div(p, factor: int):
        result = ''
        for i in range(len(p.value)):
            pv = int(p.value[i])
            result += str(int(pv / factor))
        return P(result)

    def mod(p, mod: int):
        result = ''
        for i in range(len(p.value)):
            pv = int(p.value[i])
            result += str(pv % mod)
        return P(result)

    def reduce(input, irreducible_p, p):
        result = input.mod(p)
        diff = len(result.value) - len(irreducible_p.value)
        if diff >= 0:
            while True:
                sx = P('1' + '0' * diff)
                ir_sx = irreducible_p * sx
                result += ir_sx
                result = result.mod(p)
                # Remove leading zero, since degree got reduced
                result.value = result.value.lstrip('0')
                diff = len(result.value) - len(irreducible_p.value)
                if diff < 0:
                    break
        return result

    def pad(p, width) -> str:
        return p.value.zfill(width)

    def abs(self):
        pv = self.value.replace('-', '')
        return P(pv)


class MulTab:  # Multiplication Table
    def __init__(self, irreducible_p, p=2):
        self.p = p
        self.e = len(irreducible_p.value) - 1
        self.irreducible_p = irreducible_p
        if self.e < 2:
            raise ValueError("e cannot be less than 2")
        else:
            self.width = self.p ** self.e
            self.values = [[P(self.to_base(0))] * self.width for w in
                           range(self.width)]  # Initialize two-dimensional array

    def calc_table(self):
        for i in range(1, self.width):
            for j in range(i, self.width):
                res = self.mul_mod(P(self.to_base(i)), P(self.to_base(j)))
                self.values[i][j] = res
                self.values[j][i] = res


```

```
    def mul_mod(self, p1, p2):
        mul_p = p1 * p2
        result = mul_p.reduce(self.irreducible_p, self.p)
        return result

    def print(self, raw=False):
        df = pd.DataFrame(self.values)
        def format(field): return int(
            self.pad(field.value), 2)  # Decimal converted
        if raw:
            def format(field): return self.pad(field.value)  # Raw
        df = df.applymap(format)
        print(df)

    def to_base(self, n):
        base = self.p
        digits = ""
        while n:
            digits = str(int(n % base)) + digits
            n //= base  # Floor division
        return self.pad(digits)

    def bin(self, number):
        return self.pad(bin(number)[2:])

    def pad(self, number):
        return number.zfill(self.e)


def exercise1():
    # Choose an e between 2 and 8
    e = 4

    mt = MulTab(P(ips[e]))
    mt.calc_table()
    mt.print()
```

Listing 1: Programmcode zur Aufgabe 1

Dokumentation

Zunächst haben wir die Klassen `P` für Polynom und `MulTab` für die Multiplikationstabelle generiert. Die Klasse Polynom beinhaltet Funktionen für Multiplikation, Modulo, Division, Addition und Subtraktion. Die Klasse `MulTab` enthält die Fuktion `calc_table()`, mit welcher die eigentliche Multiplikationstabelle berechnet wird. Ein Wert dieser Tabelle wird berechnet, indem der jeweilige Spaltenindex in Polynomform mit dem jeweiligen Zeilenindex in Polynomform multipliziert wird. Da die errechnete Tabelle diagonal gespiegelt werden kann, berechnet die Funktion nur eine Seite der Tabelle und spiegelt diese. Dies führt zu einer verbesserten Performance, hinsichtlich der Laufzeit.

Die Klasse `MulTab` erwartet als Übergabe ein irreduzibles Polynom. Wir haben eine Liste von diesen Polynomen erstellt und je nach gewähltem `e`, wird das passende Polynom übergeben. Daraufhin muss nur die Funktion `calc_table()` aufgerufen werden und die Multiplikationstabelle wird generiert. Zur Ausgabe der Tabelle verfügt die `MulTab` Klasse zusätzlich noch über eine `.print()` Methode. Diese ermöglicht auch das einfache Umschalten zwischen Dezimal- und Polynomdarstellung.

Programmausgaben

```
   0  1  2  3
0  0  0  0  0
1  0  1  2  3
2  0  2  3  1
3  0  3  1  2
```

Listing 2: Programmausgabe zur Aufgabe 1 mit `e=2`

```
     0    1    2    3    4    5    6    7
0  000  000  000  000  000  000  000  000
1  000  001  010  011  100  101  110  111
2  000  010  100  110  101  111  001  011
3  000  011  110  101  001  010  111  100
4  000  100  101  001  111  011  010  110
5  000  101  111  010  011  110  100  001
6  000  110  001  111  010  100  011  101
7  000  111  011  100  110  001  101  010
```

Listing 3: Programmausgabe zur Aufgabe 1 mit `e=3` und **Polynomdarstellung**

```
       0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
0      0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
1      0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
2      0   2   4   6   8  10  12  14   9  11  13  15   1   3   5   7
3      0   3   6   5  12  15  10   9   1   2   7   4  13  14  11   8
4      0   4   8  12   9  13   1   5  11  15   3   7   2   6  10  14
5      0   5  10  15  13   8   7   2   3   6   9  12  14  11   4   1
6      0   6  12  10   1   7  13  11   2   4  14   8   3   5  15   9
7      0   7  14   9   5   2  11  12  10  13   4   3  15   8   1   6
8      0   8   9   1  11   3   2  10  15   7   6  14   4  12  13   5
9      0   9  11   2  15   6   4  13   7  14  12   5   8   1   3  10
10     0  10  13   7   3   9  14   4   6  12  11   1   5  15   8   2
11     0  11  15   4   7  12   8   3  14   5   1  10   9   2   6  13
12     0  12   1  13   2  14   3  15   4   8   5   9   6  10   7  11
13     0  13   3  14   6  11   5   8  12   1  15   2  10   7   9   4
14     0  14   5  11  10   4  15   1  13   3   8   6   7   9   2  12
15     0  15   7   8  14   1   9   6   5  10   2  13  11   4  12   3
```

Listing 4: Programmausgabe zur Aufgabe 1 mit `e=4`

Lösung 2: Erweiterter Euklidischer Algorithmus

Pseudocode

1. Eingabe: Zahl `e` für die gilt, $2 \leq e \leq 8$
2. Passendes irreduziebles Polynom wählen
3. Für alle Polynome von 1 bis 2^e EEA anwenden

Programmcode und Dokumentation

Programmcode

```python
# Extended Euclidean Algorithm
def eea(p1, p2, irreducible_p, p):
    if p1 == P("0"):
        return p2, P("0"), P("1")
    gcd, u, v = eea(p2 % p1, p1, irreducible_p, p)
    pfdivmul = ((p2 // p1) * u).reduce(irreducible_p, p)
    x = (v - pfdivmul).abs()
    y = u
    return gcd, x, y

def exercise2():
    # Choose an e between 2 and 8
    e = 4

    mt = MulTab(P(ips[e]))
    df = pd.DataFrame()
    df.index = ['Field element', 'GDC', 'u', 'v', 'mul result']

    for i in range(1, 2 ** e):
        p1 = P(bin(i)[2:])
        gcd, u, v = eea(p1, mt.irreducible_p, mt.irreducible_p, mt.p)
        mul_r = mt.mul_mod(p1, u)
        result = [p1.value, gcd.value, u.value, v.value, mul_r.value]
        df = df.assign(**{str(i): result})

    print(df)
```

Listing 5: Programmcode zur Aufgabe 2

Dokumentation

Aufgabe 2 kann sich einige Funktionen und Bestandteile aus Aufgabe 1 zunutze machen. Dazu gehört in großem Umfang die Klasse `P` zum vereinfachten Rechnen mit Polynomen. Außerdem wird auch hier die `MultTab` Klasse verwendet, allerdings ohne die eigentliche Berechnung der Multiplikationstabelle auszuführen. Diese dient nur um Zugriff auf die `mul_mod` Methode zu haben.

Eine Besonderheit dieser Implementierung ist, dass die Funktion `eea` auf einer Rekursion basiert, wofür sich der EEA sehr gut eignet. Für eine optimierte Darstellung der Ergebnisse wird noch ein Pandas Dataframe verwendet. Dieses wird mit den Rückgabewerten, also den Ergebnissen des EEA, befüllt. Hierdurch können auch für größer gewählte `e`'s die jeweiligen Resultate kompakt in Form einer Tabelle ausgegeben werden.

Programmausgaben

```
                1   2   3
Field element   1  10  11
GDC             1   1   1
u               1  11  10
v               0   1   1
mul result      1   1   1
```

Listing 6: Programmausgabe zur Aufgabe 2 mit `e=2`

```
                1    2    3    4    5    6    7
Field element   1   10   11  100  101  110  111
GDC             1    1    1    1    1    1    1
u               1  110  100   11  111   10  101
v               0    1    1    1   10    1   10
mul result      1    1    1    1    1    1    1
```

Listing 7: Programmausgabe zur Aufgabe 2 mit `e=3`

Lösung 3: Linearer-Code

Pseudocode

1. Eingabe: Generatormatrix GF(2^e)
2. Mittels Gauss Zeilenstufenform errechnen $\rightarrow$ kanonische Generatormatrix erhalten
3. Einheitsmatrix E von kanonischer Generatormatrix abschneiden
4. Erhaltener Rest P transponieren und negieren (da in GF(2^e) kein negieren notwendig. Zu sich selbst invers)
5. Erhaltene transponierte Matrix um passende Einheitsmatrix ergänzen $\rightarrow$ Kontrollmatrix erhalten
6. Mittels Fehlertypen Syndrome errechnen.
   a) Fehlertyp 0^n immer auf Syndrom 0^n abbilden
   b) Syndrome für Fehlertyp die einen Fehlerbit beinhalten berechnen
   c) Fehlende Syndromwerte errechnen für mehr als ein Fehler pro Nachricht
7. Empfangene Nachricht mit Kontrollmatrix verrechnen und Syndrom ermitteln. Nachricht korrigieren über Syndromtabelle.

Programmcode und Dokumentation

Programmcode

```
def dec_to_bin(e, dec_pol):
    binary = ""
    for d in dec_pol.value:
        b = bin(int(d))[2:].zfill(e)
        binary += b
    return P(binary)


def dec_array_to_bin_array(e, dec_array):
    bin_array = []
    for d in dec_array:
        b = dec_to_bin(e, d)
        bin_array.append(b)
    return bin_array



```

```python
def bin_to_dec(e, bin_pol):
    dec = ""
    for i in range(0, len(bin_pol.value), e):
        dec += str(int(bin_pol.value[i:i+e], 2))
    return P(dec)


def bin_array_to_dec_array(e, bin_array):
    dec_array = []
    for p in bin_array:
        d = bin_to_dec(e, p)
        dec_array.append(d)
    return dec_array


# Erstellen von Einheitsmatrix
def gen_em(rows: int):
    result = []
    for i in range(rows):
        result.append(bin(2**(rows-i-1))[2:].zfill(rows))
    return result


# Erhaelt Matrix und berechnet transponierte Matrix
def gen_transposed_matrix(m):
    t_matrix = []
    for i in range(len(m[0].value)):
        pol_str = ""
        for j in range(len(m)):
            pol_str += m[j].value[i]

        t_matrix.append(P(pol_str))

    return t_matrix


# Erhaelt als Eingabe eine Matrix und Wert fuer Modulo Berechnungen.
# Berechnet Zeilenstufenform mittels Gauss von Eingabematrix.
def generate_reducedRowEchelonForm(M, e):
    if not M:
        return
    lead = 0
    rowCount = len(M)
    columnCount = len(M[0].value)
    for r in range(rowCount):
        if lead >= columnCount:
            return M
        i = r

        while M[i].value[lead] == "0":
            i += 1
            if i == rowCount:
                i = r
                lead += 1
                if columnCount == lead:
                    return M
        M[i], M[r] = M[r], M[i]
        lv = M[r].value[lead]
```

```
        M[r] = P(''.join([str(abs(int(int(mrx) / float(lv))))
                          for mrx in M[r].value])).mod(e)
        for i in range(rowCount):
            if i != r:
                lv = M[i].value[lead]
                M[i] = P(''.join([str(abs(int(iv) - int(lv) * int(rv)))
                                  for rv, iv in zip(M[r].value, M[i].value)])).mod(e)
        lead += 1
    return M


# Erstellt kanonische Generatormatrix. e ist Wert mit dem Modulo gerechnet wird.
def generate_canonical_generator_matrix(M, e):
    rref = generate_reducedRowEchelonForm(M, e)
    result = []
    for i in rref:
        if '1' in i.value:
            result.append(i)
    return result


# Erstellt Kontrollmatrix von uebergebener Matrix
def generate_control_matrix(gm):
    g = gm.copy()
    rowCount = len(g)

    for i in range(rowCount):
        g[i] = P(g[i].value[rowCount:])

    p_transposed = gen_transposed_matrix(g)

    em = gen_em(len(p_transposed))

    h = []
    for i in range(len(p_transposed)):
        h.append(P(p_transposed[i].value + em[i]))

    km = gen_transposed_matrix(h)

    return km


# Erstellt mithilfe der Kontrollmatrix eine Syndromtabelle
def generate_syndrom_table(km, e, mt):
    n = len(km)
    q = 2**e
    syndrom_table = {}

    syndrom_table['0'*len(km[0].value)] = P('0'*n)

    # generate syndroms with only 1 error
    for i in range(n):
        for j in range(1, q):
            cur_pol = P((str(j) + ('0' * i)).zfill(n))
            syndrom = ""
            for k in range(len(km[0])):
                teil_syndrom = str(mt.values[int(km[n-1-i].value[k])][j])
                teil_syndrom_dec = int(teil_syndrom, 2)
```

```
                syndrom += str(teil_syndrom_dec)

            syndrom_table[syndrom] = cur_pol

    # generate remaining errors
    for error_count in range(2, n + 1):
        for error_index_list in itertools.combinations(list(range(n)), error_count):
            for error_value_list in itertools.product(list(range(1, q)), repeat=error_count):
                error_string = "0" * n
                for error_value_pos, error_index in enumerate(error_index_list):
                    error_string = error_string[:error_index] + str(
                        error_value_list[error_value_pos]) + error_string[error_index + 1:]

                cur_pol_dec = P(error_string[::-1])
                temp_pol = P('0'*len(km[0].value * e))

                for j in range(n):
                    if cur_pol_dec.value[j] != '0':
                        syndrom = ""
                        for k in range(len(km[0])):
                            teil_syndrom = str(
                                mt.values[int(km[j].value[k])][int(cur_pol_dec.value[j])]).zfill(e)
                            syndrom += str(teil_syndrom)

                        temp_pol = (temp_pol + P(syndrom))

                temp_pol = temp_pol.mod(2)
                temp_pol_dec = bin_to_dec(e, temp_pol)
                if temp_pol_dec.value not in syndrom_table:
                    syndrom_table[temp_pol_dec.value] = P(cur_pol_dec)

    return syndrom_table


# Erhaelt als Eingabe ein empfangene Nachricht.
# Mithilfe der Kontrollmatrix wird Fehlerklasse/ Syndrom berechnet.
# Ueber diese kann in Syndromtabelle Fehler nachgeschaut werden
# und Nachricht wird korrigiert.
def error_correction_with_syndrom_table(code_polynom, km, syndrom_table):
    n = len(km)

    syndrom_class = P('0'*len(km[0].value))
    for j in range(n):
        if code_polynom.value[j] == '0':
            continue

        syndrom_class += km[j]

    syndrom_class = syndrom_class.mod(2).value
    error_polynom = syndrom_table[syndrom_class]

    return syndrom_class, (code_polynom + error_polynom).mod(2)
```

```
def calc_g_mul_ht(gm, km, mt):
    n = len(gm[0].value)

    temp = P('0' * len(km[0].value))
    for e_gm in gm:
        for j in range(n):
            t = e_gm.value[j]
            if e_gm.value[j] == '0':
                continue

            syndrom = ""
            for k in range(len(km[0])):
                teil_syndrom = str(
                    mt.values[int(km[j].value[k])][int(e_gm.value[j])])
                teil_syndrom_dec = int(teil_syndrom, 2)
                syndrom += str(teil_syndrom_dec)

            temp = (temp + P(syndrom))

    result = temp.mod(2).value

    return P(result)


def exercise3():
    # Choose an e between 2 and 8
    e = 2

    # Choose generator matrix
    gm = [
        P("10111"),
        P("01123")
    ]

    # Choose received codeword
    codeword = P("10211")

    n = len(gm[0])
    gm = dec_array_to_bin_array(e, gm)
    codeword = dec_to_bin(e, codeword)

    kgm = generate_canonical_generator_matrix(gm, 2)
    dec_kgm = bin_array_to_dec_array(e, kgm)

    km = generate_control_matrix(dec_kgm)
    mt = MulTab(P(ips[e]))
    mt.calc_table()

    syndrom_table = generate_syndrom_table(km, e, mt)
    syndrom_class, corrected_codeword = error_correction_with_syndrom_table(codeword,
    km, syndrom_table)
    g_mul_ht_result = calc_g_mul_ht(bin_array_to_dec_array(e, gm), km, mt)
```

Listing 8: Programmcode zur Aufgabe 3

Dokumentation

Zu beachten ist, dass in dieser Implementierung kein Tauschen von Spalten vorgesehen ist. Wenn spalten getauscht werden würden, würde ein äquivalenter Code `C’` im Bezug auf die Minimaldistanz erstellt werden. Jedoch ist der neu generierte Code `C’` ein anderer als `C`. Somit kann nicht immer eine systematische Generatormatrix erstellt werden (Basisspalten kompakt vorne in der Matrix).

Für Einträge der Syndromtabelle, bei welchen das Fehlerpolynom mehr als einen Fehler aufweist, kann keine eindeutige Fehlerkorrektur durchgeführt werden. Bei der Erstellung der Fehlerpolynome wird darauf geachtet, dass erst alle Polynome mit 2 Fehlern, dann mit 3 Fehlern usw. erstellt werden. Jedoch kann durch eine andere Reihenfolge der Erstellung eine unterschiedliche Syndromtabelle errechnet werden.

Programmausgaben

```
GF(2^2)^5 = GF(4)^5

Generator-Matrix:
1 0 1 1 1
0 1 1 2 3

Generator-Matrix (Binaer):
0 1 0 0 0 1 0 1 0 1
0 0 0 1 0 1 1 0 1 1

Kanonische-Generator-Matrix:
1 0 1 1 1
0 1 1 2 3

Kanonische-Generator-Matrix (Binaer):
0 1 0 0 0 1 0 1 0 1
0 0 0 1 0 1 1 0 1 1

Kontroll-Matrix:
1 1 1
1 2 3
1 0 0
0 1 0
0 0 1

Syndrom Tabelle:
Syndr.  Error
000     00000
001     00001
002     00002
003     00003
010     00010
020     00020
```

```
030      00030
100      00100
200      00200
300      00300
123      01000
231      02000
312      03000
111      10000
222      20000
333      30000
011      00011
021      00021
031      00031
012      00012
...      ...
...      ...
...      ...
320      00320
130      00130
133      01010
211      02020
322      03030
323      30010
131      10020
212      20030
321      00321
132      00132
213      00213

Syndrom Klasse:                  100
Empfangenes Codeword:          10211 (Binaer: 0100100101)
Korrigiertes Codeword:         10201 (Binaer: 0100100001)
G * Ht:                           00 (Binaer: 000)
```

Listing 9: Programmausgabe zur Aufgabe 3

Lösung 4: Hamming-Code

Pseudocode

1. Eingabe: Zahl `m` ≥ 3
2. Empfangenes Codewort/Polynom wählen mit Länge $2^m - 1$
3. Aus Eingabe `m` Kontrollmatrix `km` generieren
   a) Zur einfacheren Erstellung, Kontrollmatrix spaltenweise generieren
   b) Alle Zahlen von 1 bis 2^m, welche keine Zweierpotenz bilden, in ein Polynom umwandeln und als neue Spalte zur Matrix hinzufügen
   c) Ans Ende der Kontrollmatrix noch die dazugehörige Einheitsmatrix anhängen
4. Aus Kontrollmatrix `km` Generatormatrix `gm` generieren
   a) Alle Spalten der Kontrollmatrix, welche keinem Einheitsvektor entsprechen, transponieren und als neue Zeile zur Generatormatrix hinzufügen
   b) An den Anfang der Generatormatrix noch die dazugehörige Einheitsmatrix anhängen
5. Mithilfe von Kontrollmatrix `km` Decodierung durchführen
   a) Kontrollmatrix transponieren
   b) $y * H^T$ berechnen
   c) Faktor `a` berechnen
   d) Fehler-Polynom bestimmen
   e) Fehler durch Addition von empfangenen Codewort/Polynom und dem Fehler-Polynom korrigieren

Programmcode und Dokumentation

Programmcode

```
def is_power_of_two(n):
    return (n != 0) and (n & (n-1) == 0)


def generate_hamming_control_matrix(m):
    control_matrix = []
    for i in range(1, 2**m):
        if not is_power_of_two(i):
            item = P(str(bin(i))[2:].zfill(m))
            control_matrix.append(item)

    for i in gen_em(m):
        control_matrix.append(P(i))

    return control_matrix


def hamming_control_matrix_to_generator_matrix(control_matrix):
    m = len(control_matrix[0].value)

    generator_matrix = []

    for i in gen_em(len(control_matrix)-m):
        generator_matrix.append(P(i))

    p_transposed = gen_transposed_matrix(
        control_matrix[:-m])

    generator_matrix += p_transposed

    return generator_matrix


def decode_hamming(codeword, control_matrix):
    cm_transposed = gen_transposed_matrix(control_matrix)

    # y * H^T berechnen
    pol_str = ""
    for i in cm_transposed:
        sum = 0
        for j in range(len(codeword.value)):
            sum += int(codeword.value[j]) * int(i.value[j])
        pol_str += str(sum)

    y_ht = P(pol_str).mod(2)

    # faktor a berechnen
    a = 0
    for i in y_ht.value:
        if int(i) > 0:
            a = int(i)
            break

```

```
    # fehler berechnen
    error = None
    for i, item in enumerate(gen_transposed_matrix(cm_transposed)):
        if item.mul(a) == y_ht:
            error_string = '0'*len(cm_transposed[0].value)
            error_string = error_string[:i] + str(a) + error_string[i+1:]
            error = P(error_string)

    if error:
        corrected_codeword = (codeword + error).mod(2)
        return corrected_codeword

    return codeword


def exercise4():
    # Choose an m greater or equal 3
    m = 3

    # Choose received codeword containing 0 and 1 with length (2^m - 1)
    codeword = P("0101111")

    km = generate_hamming_control_matrix(m)
    gm = hamming_control_matrix_to_generator_matrix(km)
    corrected_codeword = decode_hamming(codeword, km)
```

Listing 10: Programmcode zur Aufgabe 4

Dokumentation

Bei der Implementierung von Aufgabe 4 kann erneut auf die bereits bestehenden Funktionen der vorhergehenden Aufgaben zurückgegriffen werden. So kommt unter anderem die Polynom Klasse `P` oder auch die Generierung von Einheitsmatrizen mittels `gen_em()` zum Einsatz. Da auch an mehreren Stelle das Transponieren von Matrizen notwendig ist, wird zudem die Funktion `gen_transposed_matrix()` miteinbezogen.

Die eigentliche Implementierung hält sich unmittelbar an den oben beschriebenen Pseudocode-Ablauf. Die einzelnen Schritte wurden in jeweils eine gesonderte Funktion ausgelagert. Durch die einfache Angabe eines Wertes für `m`, kann so bereits mit jeweils nur einem Funktionsaufruf die benötigte Kontrollmatrix, sowie die Generatormatrix generiert werden. Die Methode `decode_hamming()` kann wiederum ein empfangenes Codewort entgegen nehmen und mithilfe der Kontrollmatrix mögliche Fehler detektieren und anschließend korrigieren. Zu beachten ist hierbei nur, dass das empfangene Codewort eine Länge von $2^m - 1$ Zeichen aufweisen muss.

Programmausgaben

```
Kontroll-Matrix:
0 1 1 1 1 0 0
1 0 1 1 0 1 0
1 1 0 1 0 0 1

Generator-Matrix:
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

Empfangenes Codeword:   0101111
Korrigiertes Codeword:  0001111
```

Listing 11: Programmausgabe zur Aufgabe 4 mit `m=3`

```
Kontroll-Matrix:
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 0 0 0 0
0 0 0 0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 0 1 0 0 0
0 1 1 1 0 0 0 1 1 1 1 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 1 0 0
1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 0 1 0
1 1 0 1 1 0 1 0 1 0 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 0 0 0 1

Generator-Matrix:
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0
0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1
0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1
0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0
0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1
0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 1
0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0
0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1
0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 1 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 1 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 1 0 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 1 0 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 1 0 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 1 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 1 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 1 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1

Empfangenes Codeword:   0000000000000000000000000101111
Korrigiertes Codeword:  0000000000000000000000000111111
```

Listing 12: Programmausgabe zur Aufgabe 4 mit `m=5`

Lösung 5: Reed-Muller-Code

Pseudocode

1. Eingabe: `r` und `m`
2. Rekursive Funktion `generate_reed_muller_code(r, m)` zum Generieren des Reed-Muller-Codes, welche `r` und `m` als Parameter erhält
   a) Wenn `r == 0`, Polynom mit 2^m 1er zurückgeben
   b) Wenn `r > m`, Ergebnis von `generate_reed_muller_code(m, m)` zurückgeben
   c) Sonst:
      i. rm_1 = `generate_reed_muller_code(r, m-1)`
      ii. rm_2 = `generate_reed_muller_code(r-1, m-1)`
      iii. Leere Matrix `rm_generator_matrix` erstellen
      iv. Schleife bis zur Anzahl der Stellen in rm_1 und jeweils zweilmal den Wert von rm_1 an der Stelle i hintereinanderschreiben und als Polynom an die Matrix `rm_generator_matrix` hinzufügen
      v. Schleife bis zur Anzahl der Stellen in rm_2 und jeweils 0en und den Wert von rm_2 an der Stelle i hintereinanderschreiben und als Polynom an die Matrix `rm_generator_matrix` hinzufügen

Programmcode und Dokumentation

Programmcode

```
def generate_reed_muller_code(r, m):
    if r == 0:
        return [P('1' * 2**m)]
    elif r > m:
        return generate_reed_muller_code(m, m)

    rm_1 = generate_reed_muller_code(r, m-1)
    rm_2 = generate_reed_muller_code(r-1, m-1)
    rm_generator_matrix = []

    for i in range(len(rm_1)):
        rm_generator_matrix.append(P(rm_1[i].value + rm_1[i].value))

    for i in range(len(rm_2)):
        rm_generator_matrix.append(P('0' * len(rm_1[0].value) + rm_2[i].value))

    return rm_generator_matrix
```

```python
def exercise5():
    # Choose r, m for Reed-Muller-Code construction
    r = 1
    m = 5

    reed_muller_code = generate_reed_muller_code(r, m)
```

Listing 13: Programmcode zur Aufgabe 5

Dokumentation

Die Funktion `generate_reed_muller_code(r, m)` nimmt die Parameter r und m an, welche vom Benutzer zu setzen sind. Die Funktion ist Rekursiv. Die Abbruchbedingung ist erreicht, wenn r = 0 ist. Hier wird ein Polynom mit 2^m 1en zurückgegeben. Andernfalls, wenn r > m ist wird die Funktion erneut aufgerufen mit generate_reed_muller_code(m, m) und die Lösung davon wird zurückgegeben. Ansonsten werden zunächst wie folgt zwei Reed-Muller-Codes generiert:

`rm_1 = generate_reed_muller_code(r, m-1)`

`rm_2 = generate_reed_muller_code(r-1, m-1)`

Danach wird rm_1 iteriert und der Wert an der Stelle i zweimal hintereinander geschrieben und an eine zuvor angelegte leere Matrix gehängt. Auch rm_2 wird iteriert und der Wert von rm_2 an der Stelle i und davor die Stellen mit 0ern aufgefüllt ebenso an die Matrix angehängt.

Programmausgaben

```
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 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 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
```

Listing 14: Programmausgabe zur Aufgabe 5 mit `r=1` und `m=5`

```
1 1 1 1 1 1 1 1
0 1 0 1 0 1 0 1
0 0 1 1 0 0 1 1
0 0 0 1 0 0 0 1
0 0 0 0 1 1 1 1
0 0 0 0 0 1 0 1
0 0 0 0 0 0 1 1
```

Listing 15: Programmausgabe zur Aufgabe 5 mit `r=2` und `m=3`

Lösung 6: Reed-Solomon-Code

Pseudocode

1. Eingabe: Parameter e für den gilt d $2 \leq d \leq 2$, Minimaldistanz d
2. Ermitteln des primitiven Elements in GF(2^e)
3. Generatorpolynom und Kontrollpolynom mit Hilfe des ermittelten primitiven Elements und der vorgegebenen Minimaldistanz berechnen
4. Durch Generatorpolynom dazugehörige Generatormatrix erstellen
5. Durch Kontrollpolynom dazugehörige Kontrollmatrix für Code RS(2^e, d) erstellen

Programmcode und Dokumentation

Programmcode

```
def determine_primitive_element(q, mt):
    gf_target = [x for x in range(1, q)]

    for alpha in range(1, q):
        gf_without_zero = []
        for i in range(q-1):  # 0 <= i <= q-2
            if i == 0:
                gf_without_zero.append(1)
                continue
            elif i == 1:
                gf_without_zero.append(alpha)
                continue

            result = P(str(bin(alpha))[2:])
            for x in range(i-1):
                result = mt.mul_mod(result, P(str(bin(alpha)[2:])))

            gf_without_zero.append(int(int(result.value, 2)))

        if set(gf_without_zero) == set(gf_target):
            return alpha
    return None

def add_with_mod(p1, p2, q):
    width = max(len(p1.value), len(p2.value))
    p1, p2 = p1.pad(width), p2.pad(width)
    result = ''
    for i in range(width):
        pv1, pv2 = int(p1[i]), int(p2[i])
        result += str((pv1 + pv2) % q)
    return P(result)
```

```python
def mul_bin(p1, p2, mt, q):
    pl1, pl2 = len(p1.value), len(p2.value)
    width = pl1 + pl2 - 1
    result = [0] * width
    for i in range(pl1):
        for j in range(pl2):
            pv1, pv2 = int(p1.value[i]), int(p2.value[j])
            r = int(mt.mul_mod(P(str(bin(pv1)[2:])), P(
                str(bin(pv2)[2:]))).value, 2)
            result[i + j] = (result[i + j] + r) % q
    result = ''.join(str(x) for x in result)  # List to string
    result = result.lstrip('0')  # Remove leading zeros
    return P(result or '0')


def generate_reed_solomon_generator_polynom(alpha, q, d, mt):
    g_list = []
    for i in range(1, d):  # 1 <= i <= d-1
        # nicht -(alpha ** i), da selbstinversiv in q=2**e
        value = P(str(bin(alpha))[2:])
        for x in range(i-1):
            value = mt.mul_mod(value, P(str(bin(alpha)[2:])))

        p = P('1' + str(int(value.value, 2)))
        g_list.append(p)

    result_polynom = P('1')
    for i in range(len(g_list)):
        result_polynom = mul_bin(result_polynom, g_list[i], mt, q)

    return result_polynom


def generate_reed_solomon_control_polynom(alpha, q, d, mt):
    g_list = [P('11')]  # (1 - alpha**0)
    for i in range(d, q-1):  # d <= i <= q-2
        # nicht -(alpha ** i), da selbstinversiv in q=2**e
        value = P(str(bin(alpha))[2:])
        for x in range(i-1):
            value = mt.mul_mod(value, P(str(bin(alpha)[2:])))

        p = P('1' + str(int(value.value, 2)))
        g_list.append(p)

    result_polynom = P('1')
    for i in range(len(g_list)):
        result_polynom = mul_bin(result_polynom, g_list[i], mt, q)

    return result_polynom


def generate_reed_solomon_control_matrix(control_polynom, d):
    H = []
    max = d-1
    for i in range(d-1):
        pol_str = ('0' * i) + control_polynom.value + ('0' * (max-i-1))
        H.append(P(pol_str))
    return H
```

```python
def generate_reed_solomon_generator_matrix(generator_polynom, q, d):
    inverted_gp_string = generator_polynom.value[::-1]
    n = q-1

    G = []
    max = n - len(inverted_gp_string) + 1
    for i in range(max):
        pol_str = ('0' * i) + inverted_gp_string + ('0' * (max-i-1))
        G.append(P(pol_str))
    return G

def generate_reed_solomon_vandermonde_matrix_old(polynom, q):
    n = len(polynom.value)
    m = n

    V = []
    for i in range(0, m):  # 0 <= i <= m-1
        p_string = ''
        for j in range(n):  # 0 <= j <= n-1
            value = (int(polynom.value[i]) ** j) % q
            p_string += str(value)
        V.append(P(p_string))

    return V

def generate_reed_solomon_vandermonde_matrix(polynom, q, mt):
    n = len(polynom.value)
    m = n

    V = []
    for i in range(0, m):  # 0 <= i <= m-1
        p_string = ''
        for j in range(n):  # 0 <= j <= n-1
            if j == 0:
                p_string += "1"
                continue

            result_polynom = P(str(polynom.value[i]))
            for k in range(j-1):
                result_polynom = mul_bin(
                    result_polynom, P(polynom.value[i]), mt, q)

            p_string += str(result_polynom.value)
        V.append(P(p_string))

    return V

def generate_reed_solomon_code(e, d, mt):
    alpha = determine_primitive_element(2**e, mt)
    q = 2**e

    generator_polynom = generate_reed_solomon_generator_polynom(alpha, q, d, mt)
    generator_matrix = generate_reed_solomon_generator_matrix(generator_polynom, q, d)
    control_polynom = generate_reed_solomon_control_polynom(alpha, q, d, mt)
    control_matrix = generate_reed_solomon_control_matrix(control_polynom, d)
    vandermonde_matrix = generate_reed_solomon_vandermonde_matrix(generator_polynom, q,
     mt)

```

```python
def exercise6():
    # Choose e, d for Reed-Solomon-Code construction with q = 2^e
    e = 3
    d = 3

    mt = MulTab(P(ips[e]))
    generate_reed_solomon_code(e, d, mt)
```

Listing 16: Programmcode zur Aufgabe 6

Dokumentation

Relevant für einen Reed Solomon Code sind die Parameter `e` und `d`. Es können hierbei Codes erzeugt werden, bei denen durch die Konstruktion die angegebene Minimaldistanz gewährleistet werden kann.
Bei dieser Aufgabe ist darauf zu achten, dass bei der Multiplikation und Addition zu Berechnungen von Polynomen in Polynomen kommen kann. Diese Verschachtelung tritt bei den beiden Funktionen `generate_reed_solomon_generator_polynom` und `generate_reed_solomon_control_polynom` auf.

Programmausgaben

```
Alpha: 2
Generator-Polynom: 165
Kontroll-Polynom: 164107

Generator-Matrix:
5 6 1 0 0 0 0
0 5 6 1 0 0 0
0 0 5 6 1 0 0
0 0 0 5 6 1 0
0 0 0 0 5 6 1

Kontroll-Matrix:
1 6 4 1 0 7 0
0 1 6 4 1 0 7

Vandermonde-Matrix:
1 1 1
1 6 3
1 5 6
```

Listing 17: Programmausgabe zur Aufgabe 6 mit `e=3` und `d=3`

```
Alpha: 2
Generator-Polynom: 153772
Kontroll-Polynom: 176

Generator-Matrix:
2 7 7 3 5 1 0
0 2 7 7 3 5 1

Kontroll-Matrix:
1 7 6 0 0 0 0
0 1 7 6 0 0 0
0 0 1 7 6 0 0
0 0 0 1 7 6 0
0 0 0 0 1 7 6

Vandermonde-Matrix:
1 1 1 1 1 1
1 5 6 4 3 2
1 3 5 2 6 7
1 7 2 3 4 6
1 7 2 3 4 6
1 2 4 5 7 3
```

Listing 18: Programmausgabe zur Aufgabe 6 mit `e=3` und `d=6`