Санкт-Петербургский Политехнический Университет Петра Великого
Институт компьютерных наук и технологий
Кафедра компьютерных систем и программных технологий

Телекоммуникационные технологии

Отчет по лабораторной работе №7
Помехоустойчивое кодирование

Работу выполнил:
Чугунов А.А.
Группа: 33501/4
Преподаватель:
Богач Н.В.

Санкт-Петербург
2017

Содержание

1. Цель и задачи

1.1. Цель работы

Изучение методов помехоустойчивого кодирования и сравнения их свойств.

1.2. Постановка задачи

Провести кодирование/декодирование сигнала, полученного с помощью функции randerr кодом Хэмминга 2-мя способами: с помощью встроенных функций encode/decode, а также через создание проверочной и генераторной матриц и вычисление синдрома. Оценить корректирующую способность кода.

Выполнить кодирование/декодирование циклическим кодом, кодом БЧХ, кодом Рида-Соломона. Оценить корректирующую способность кода.

2. Теоретическая информация

2.1. Кодирование

Физическое кодирование — линейное преобразование двоичных данных, осуществляемое для их передачи по физическому каналу (такому как оптическое волокно или витая пара). Физическое кодирование может менять форму, ширину полосы частот и гармонический состав сигнала в целях осуществления синхронизации приёмника и передатчика, устранения постоянной составляющей или уменьшения аппаратных затрат.

Обнаружение ошибок в технике связи — действие, направленное на контроль целостности данных при записи/воспроизведении информации или при её передаче по линиям связи. Исправление ошибок (коррекция ошибок) — процедура восстановления информации после чтения её из устройства хранения или канала связи.

Для обнаружения ошибок используют коды обнаружения ошибок, для исправления — корректирующие коды (коды, исправляющие ошибки, коды с коррекцией ошибок, помехоустойчивые коды).

2.2. Типы помехоустойчивого кодирования

2.2.1. Кодирование Хэмминга

Коды Хемминга — простейшие линейные коды с минимальным расстоянием 3, то есть способные исправить одну ошибку. Код Хемминга может быть представлен в таком виде, что синдром

$$\vec{s} = \vec{r}H^T \tag{1}$$

Это принятый вектор, будет равен номеру позиции, в которой произошла ошибка. Это свойство позволяет сделать декодирование очень простым.

Коды Хэмминга являются самоконтролирующимися кодами, то есть кодами, позволяющими автоматически обнаруживать ошибки при передаче данных.

Для построения самокорректирующегося кода, рассчитанного на исправление одиночных ошибок, одного контрольного разряда недостаточно. Как видно из дальнейшего, количество контрольных разрядов k должно быть выбрано так, чтобы удовлетворялось неравенство

$$2^k \geq k + m + 1 \tag{2}$$

или

$$k \geq \log_2(k + m + 1) \tag{3}$$

где m — количество основных двоичных разрядов кодового слова.

Минимальные значения k при заданных значениях m, найденные в соответствии с этим неравенством, приведены в таблице.

Диапазон m	k_{min}
1	2
2-4	3
5-11	4
12-26	5
27-57	6

Рис. 2.2.1. Значения K_{min} в зависимости от m.

Построение кодов Хэмминга основано на принципе проверки на четность числа единичных символов: к последовательности добавляется такой элемент, чтобы число единичных символов в получившейся последовательности было четным.

$$r_1 = i_1 \oplus i_2 \oplus ... \oplus i_k \tag{4}$$

$$S = i_1 \oplus i_2 \oplus ... \oplus i_n \oplus r_1 \tag{5}$$

Тогда если $S = 0$ - ошибки нет, иначе есть однократная ошибка.

Такой код называется $(k + 1, k)$ или $(n, n - 1)$. Первое число — количество элементов последовательности, второе — количество информационных символов.

Для примера рассмотрим классический код Хемминга $(7, 4)$. Сгруппируем проверочные символы следующим образом:

$$r_1 = i_1 \oplus i_2 \oplus i_3 r_2 = i_2 \oplus i_3 \oplus i_4 r_3 = i_1 \oplus i_2 \oplus i_4 \tag{6}$$

Получение кодового слова выглядит следующим образом:

$$(i_1\ i_2\ i_3\ i_4) \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{pmatrix} = (i_1\ i_2\ i_3\ i_4\ r_1\ r_2\ r_3) \tag{7}$$

На вход декодера поступает кодовое слово $V = (i'_1, i'_2, i'_3, i'_4, r'_1, r'_2, r'_3)$ где штрихом помечены символы, которые могут исказиться в результате помехи. В декодере в режиме исправления ошибок строится последовательность синдромов:

$S_1 = r_1 \oplus i_1 \oplus i_2 \oplus i_3$

$S_2 = r_2 \oplus i_2 \oplus i_3 \oplus i_4$

$S_3 = r_3 \oplus i_1 \oplus i_2 \oplus i_4$

$S = (S_1, S_2, S_3)$ называется синдромом последовательности.

Получение синдрома выглядит следующим образом:

$$(i_1\, i_2\, i_3\, i_4\, r_1\, r_2\, r_3) \begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} S_1 & S_2 & S_3 \end{pmatrix} \tag{8}$$

2.2.2. Циклические коды

Циклический код — линейный код, обладающий свойством цикличности, то есть каждая циклическая перестановка кодового слова также является кодовым словом. Используется для преобразования информации для защиты её от ошибок.

2.2.3. Коды Боуза-Чоудхури-Хоквингема (БЧХ)

Коды Боуза — Чоудхури — Хоквингема (БЧХ-коды) — в теории кодирования это широкий класс циклических кодов, применяемых для защиты информации от ошибок. Отличается возможностью построения кода с заранее определёнными корректирующими свойствами, а именно, минимальным кодовым расстоянием. Частным случаем БЧХ-кодов является код Рида — Соломона.

2.2.4. Коды Рида-Соломона

Коды Рида — Соломона (англ. Reed–Solomon codes) — недвоичные циклические коды, позволяющие исправлять ошибки в блоках данных. Элементами кодового вектора являются не биты, а группы битов (блоки).

Код Рида — Соломона является частным случаем БЧХ-кода.

3. Ход работы

Реализация различных типов кодирования с помощью Python:

Листинг 1: Код в Python, Hamming Code

```
1  import math
2
3
4  # several utility functions.
5  #
6  def isPowerOfTwo(n):
7      m = math.log(n, 2)
8      return m == int(m)
9
10
11 def power2Below(n):
12     return 2 ** int(math.log(n, 2))
13
14
15 # Is the parity bit in position i, a parity bit for position j?
16 # NB that >> is the shift-right operator.
17 #
18 def covers(i, j):
```

```
19      return (j >> int(math.log(i, 2))) % 2 == 1
20
21
22  # sum all of the bits covered by parity bit i.
23  #
24  def sumBits(bits, i, j):
25      if j > len(bits):
26          return 0
27      else:
28          restAnswer = sumBits(bits, i, j + 1)
29          if covers(i, j):
30              # NB: j-1 below because lists are 0-based.
31              #
32              return bits[j - 1] + restAnswer
33          else:
34              return restAnswer
35
36
37  def hasOddParity(bits, i):
38      return sumBits(bits, i, i) % 2 == 1
39
40
41  def hasEvenParity(bits, i):
42      return not (hasOddParity(bits, i))
43
44
45  # constructs an integer from a list of bits.
46  #
47  def bitsToNumber(bits):
48      if bits == []:
49          return 0
50      else:
51          n = bits[0] * (2 ** (len(bits) - 1))
52          return n + bitsToNumber(bits[1:])
53
54
55  # prepare accepts the list of data bits and an index
56  # into the list (originally 1) and returns a new list with
57  # zeros in the power of two positions in the list.
58  #
59  def prepare(bits, i):
60      if bits == []:
61          return []
62      else:
63          if isPowerOfTwo(i):
64              return [0] + prepare(bits, i + 1)
65          else:
66              return [bits[0]] + prepare(bits[1:], i + 1)
67
68
69  # The input list has zeros in the parity positions; the
70  # output has the parity bits properly set.
71  #
72  def setParityBits(bits, i):
73      if i > len(bits):
74          return []
75      else:
76          restAnswer = setParityBits(bits, i + 1)
77          if isPowerOfTwo(i):
78              if hasOddParity(bits, i):
```

```
79                  return [0] + restAnswer
80              else:
81                  return [1] + restAnswer
82          else:
83              return [bits[i - 1]] + restAnswer
84
85
86  def encode(bits):
87      paritysAreZero = prepare(bits, 1)
88      return setParityBits(paritysAreZero, 1)
89
90
91  def decode(bits):
92      parityResults = checkParity(bits, power2Below(len(bits)))
93      n = bitsToNumber(parityResults)
94
95      if n != 0:
96          print("NB: bit ", n, " is bad. Flipping.")
97
98          # WARNING!!! Destructive update!
99          #
100         bits[n - 1] = 1 - bits[n - 1]
101
102     return extractData(bits, 1)
103
104
105 # extractData gathers the bits in non-parity positions.
106 #
107 def extractData(bits, i):
108     if i > len(bits):
109         return []
110     else:
111         restAnswer = extractData(bits, i + 1)
112         if isPowerOfTwo(i):
113             return restAnswer
114         else:
115             return [bits[i - 1]] + restAnswer
116
117
118 # i is a power of 2. checkParity works from right-to-left to
119 # simplify construction of the correct result bit list.
120 #
121 def checkParity(bits, i):
122     if i == 1:
123         return [0] if hasOddParity(bits, i) else [1]
124     else:
125         bit = 0 if hasOddParity(bits, i) else 1
126         return [bit] + checkParity(bits, power2Below(i - 1))
127
128
129 bits = [1, 1, 1, 1]
130 print("input data:" + str(bits))
131 encoded = encode(bits)
132 print("encoded data:" + str(encoded))
133
134 decoded = decode(encoded)
135 print("decoded data:" + str(decoded))
136
137 wrongData = [1, 0, 1, 0, 1, 1, 1]
138 print("badly encoded data:" + str(wrongData))
```

```
decode(wrongData)
```

Листинг 2: Код в Python, Hamming Code Matrix

```
import random
# the encoding matrix
G = ['1101', '1011', '1000', '0111', '0100', '0010', '0001']
# the parity-check matrix
H = ['1010101', '0110011', '0001111']
Ht = ['100', '010', '110', '001', '101', '011', '111']
# the decoding matrix
R = ['0010000', '0000100', '0000010', '0000001']

# p = ''.join([random.choice('01') for k in range(4)])
p = '1111'
print ('Input bit string: ' + p)

x = ''.join([str(bin(int(i, 2) & int(p, 2)).count('1') % 2) for i in G])
print ('Encoded bit string to send: ' + x)

# add 1 bit error
e = random.randint(0, 7)
# counted from left starting from 1
print ('Which bit got error during transmission (0: no error): ' + str(e))
if e > 0:
    x = list(x)
    x[e - 1] = str(1 - int(x[e - 1]))
    x = ''.join(x)
print ('Encoded bit string that got error during tranmission: ' + x)

z = ''.join([str(bin(int(j, 2) & int(x, 2)).count('1') % 2) for j in H])
if int(z, 2) > 0:
    e = int(Ht[int(z, 2) - 1], 2)
else:
    e = 0
print ('Which bit found to have error (0: no error): ' + str(e))

# correct the error
if e > 0:
    x = list(x)
    x[e - 1] = str(1 - int(x[e - 1]))
    x = ''.join(x)

p = ''.join([str(bin(int(k, 2) & int(x, 2)).count('1') % 2) for k in R])
print('Corrected output bit string: ' + p)
```

Листинг 3: Код в Python,Cyclic Coding

```
def crc(msg, div, code='000'):
    """Cyclic Redundancy Check
    Generates an error detecting code based on an inputted message
    and divisor in the form of a polynomial representation.
    Arguments:
        msg: The input message of which to generate the output code.
        div: The divisor in polynomial form. For example, if the polynomial
            of x^3 + x + 1 is given, this should be represented as '1011' in
            the div argument.
        code: This is an option argument where a previously generated code may
            be passed in. This can be used to check validity. If the inputted
            code produces an outputted code of all zeros, then the message has
            no errors.
```

```
    Returns:
        An error-detecting code generated by the message and the given divisor.
    """
    # Append the code to the message. If no code is given, default to '000'
    msg = msg + code

    # Convert msg and div into list form for easier handling
    msg = list(msg)
    div = list(div)

    # Loop over every message bit (minus the appended code)
    for i in range(len(msg)-len(code)):
        # If that messsage bit is 1, perform modulo 2 multiplication
        if msg[i] == '1':
            for j in range(len(div)):
                # Perform modulo 2 multiplication on each index of the divisor
                msg[i+j] = str((int(msg[i+j])+int(div[j]))%2)

    # Output the last error-checking code portion of the message generated
    return ''.join(msg[-len(code):])


# TEST 1 ##################################################################################
print('Test 1 ----------------------------------')
# Use a divisor that simulates: x^3 + x + 1
div = '1011'
msg = '11010011101101'

print('Input message:', msg)
print('Divisor:', div)

# Enter the message and divisor to calculate the error-checking code
code = crc(msg, div)

print('Output code:', code)

# Perform a test to check that the code, when run back through, returns an
# output code of '000' proving that the function worked correctly
print('Success:', crc(msg, div, code) == '000')


# TEST 2 ##################################################################################
print('Test 2 ----------------------------------')
# Use a divisor that simulates: x^2 + 1
div = '0101'
msg = '0110'

print('Input message:', msg)
print('Divisor:', div)

# Enter the message and divisor to calculate the error-checking code
code = crc(msg, div)

print('Output code:', code)

# Perform a test to check that the code, when run back through, returns an
# output code of '000' proving that the function worked correctly
print('Success:', crc(msg, div, code) == '000')
```

Для рассмотрения кодов Рида-Соломона и БЧХ-кода был использован Matlab.

3.1. Коды Хэмминга

Ниже представлены результаты работы кодов Хэмминга, реализацию которых пришлось написать самостоятельно.

```
input data:[1, 1, 1, 1]
encoded data:[0, 0, 1, 0, 1, 1, 1]
decoded data:[1, 1, 1, 1]
badly encoded data:[1, 0, 1, 0, 1, 1, 1]
NB: bit  1  is bad. Flipping.
```

Рис. 3.1.1. Исходное сообщение и его код Хэмминга.

Наблюдаем, как с помощью кодов мы смогли найти ошибку.

```
Input bit string: 1111
Encoded bit string to send: 1111111
Which bit got error during transmission (0: no error): 6
Encoded bit string that got error during tranmission: 1111101
Which bit found to have error (0: no error): 6
Corrected output bit string: 1111
```

Рис. 3.1.2. Коды Хэмминга матричный способ.

Аналогичная ситуация с матричным способом.

3.2. Циклические коды

Реализацию циклического кода тоже пришлось написать самостоятельно. В данном случае, после кодирование получаем специальный трехзначный код, с помощью которого можем проверить, верно ли передано сообщение. В соответствии с этим, получаем вывод success, если посылка прошла удачно.

```
Test 1 --------------------------
Input message: 11010011101101
Divisor: 1011
Output code: 111
Success: True
Test 2 --------------------------
Input message: 0110
Divisor: 0101
Output code: 100
Success: True
```

Рис. 3.2.1. Исходное сообщение и его циклический код.

3.3. Коды Боуза-Чоудхури-Хоквингема (БЧХ)

Для кодирования/декодирования с помощью кодов БЧХ использовались, соответственно, функции bchenc/bchdec. При кодировании сообщений с кодовым расстоянием, равным 1, получали, как пример, закодированные сообщения с кодовым расстоянием равным 3, или 4.

Массивы после кодирования и декодирования представлены на рис. 3.3.1 и 3.3.2.

```
code = GF(2) array.

Array elements =

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

Рис. 3.3.1. Массив после кодирования

```
dcode = GF(2) array.

Array elements =

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

Рис. 3.3.2. Массив после декодирования

3.4. Коды Рида-Соломона

При использовании кодов Рида-Соломона в виде стандартной функции rsenc можно наблюдать вектор cnumerr, который содержит количества исправляемых ошибок.

```
cnumerr =

     1
     2
    -1
```

Рис. 3.4.1. Количество исправляемых ошибок cnumerr.

При кодировании сообщений с кодовым расстоянием, равным 1, получали, как пример, закодированные сообщения с кодовым расстоянием равным 3, или 4.

4. Выводы

Кодирование - важный процесс при передаче сигналов по каналам связи. Методы кодирования дополняют методы модуляции для обеспечения улучшения качества передачи, для предотвращения ошибок при передаче, а также защищенности данных от получения их другими лицами. Рассмотрены различные методы кодирования, которые являются самокорректирующимися: коды Хэмминга, циклические коды, коды Боуза-Чоудхури-Хоквингема, коды Рида-Соломона.