МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования

«МИРЭА – Российский технологический университет»
РТУ МИРЭА

Институт искусственного интеллекта
Кафедра информационной безопасности (БК №252)

Курсовая работа
тема:
«Код БЧХ»

Студент группы ККСО-01-19: *Алабин А. Ю.*

Руководитель работы: *преподаватель Чернышев Н. Н.*

Москва 2022

Оглавление

Введение

В данной работе рассматривается помехоустойчивое кодирование БЧХ: алгоритмы построения кода, кодирования и декодирования сообщений, оценка эффективности кода.

Помехоустойчивые коды, в частности БЧХ, нашли своё применение в международных стандартах передачи данных, например, в Ethernet.

Теоретическая часть

Блочный код – код, к/с(кодовые слова) которого имеют одинаковую длину n при длине информационных блоков $k < n$.

Линейный код – блочный код, к/с которого представляются векторами линейного k-мерного подпространства(n-мерного пространства), т.е.

$$v(x) = c_1\overline{x_1} + \cdots + c_k\overline{x_k},$$

где $\overline{x_i} = (a_{i1}, \ \ldots, \ a_{in})$ – один из k базисных векторов линейного k-мерного подпр-ва.

Минимальное кодовое расстояние d_{min} – минимальное число попарно отличающихся разрядов среди всех пар кодовых слов, можно определить, как минимальный вес(по Хэммингу) кодового слова.

Известно равенство, определяющее число ошибок t, которые может исправить линейный код:

$$t = \lfloor (d_{min} - 1)/2 \rfloor$$

Циклический код – линейный код, каждое к/с которого можно получить за некоторое количество циклических сдвигов другого к/с.

Циклический сдвиг на j разрядов удобно описать, если воспринимать к/с v, как полином $v(x)$, где каждый символ к/с - это коэффициент при соответствующей степени x:

$$ROT(v, j): \ x^j \, v(x) \ mod \ (x^n - 1)$$

Код БЧХ (Боуза-Чоудхури-Хоквингема) – циклический код, задающийся порождающим полиномом g(x) и гарантирующий заданное минимальное расстояние к/с (а значит и число исправляемых ошибок t – согласно равенству выше);
для поиска g(x) необходимо задать n – длину кодового блока и минимальное кодовое расстояние(подробнее – см. Кодирование III):

$$d_{min} \geq \ \delta = 2t + 1$$

Порождающий полином g(x) – имеет корнями $\alpha^1, \ldots, \alpha^{\delta-1}$ и при этом он минимальной степени, где α – примитивный элемент из $GF(2^m), \ m\colon n = 2^m - 1$

Код БЧХ имеет параметры:

- n – величина кодового блока (задаётся – в битах)
- t – число исправляемых ошибок (задаётся – в битах)
- k – величина информационного блока (считается – см. Кодирование III)

Также для оценки эффективности кода вводятся параметры:

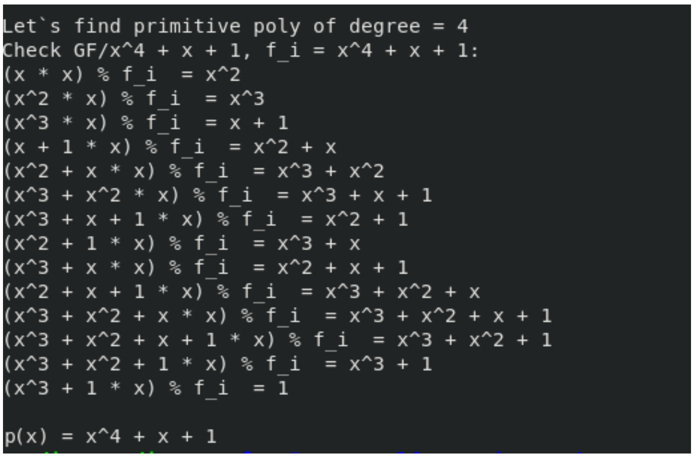
- Rate – скорость передачи, $Rate = n/k$
- Safety – процент гарантированно верно переданных бит $Safety = t/n$

Описание алгоритмов кодирования и декодирования

Кодирование:

I. *Задание кода:*
- величина передаваемого(кодированного) блока данных $= 2^m - 1 = |GF(2^m)/\{0\}|$
- число исправляемых ошибок t

II. *Построение $GF(2^m)$:*
В начале нужно построить $GF(2^m)/\{0\}$ – мультипликативную группу поля(далее обозначаемую просто $GF(2^m)$):
$GF(2^m) \cong GF(2)/p(x)$ – представление в виде
где $p(x)$ – неприводимый многочлен,
тогда $GF(2)/p(x) = \{\ \alpha^k \ mod\ p(x)\ \mid\ k \in \overline{0, m-1}\ \}$
где α – примитивный элемент, т.е. его степени порождают всё $GF(2^m)$
в нашем случае все выбираемые $p(x)$: $\alpha = x$ – примитивный элемент.
Пример поиска $p(x)$ степени 4 (рассматриваем только неприводимые мн-ны):



```
Let`s find primitive poly of degree = 4
Check GF/x^4 + x + 1, f_i = x^4 + x + 1:
(x * x) % f_i  = x^2
(x^2 * x) % f_i  = x^3
(x^3 * x) % f_i  = x + 1
(x + 1 * x) % f_i  = x^2 + x
(x^2 + x * x) % f_i  = x^3 + x^2
(x^3 + x^2 * x) % f_i  = x^3 + x + 1
(x^3 + x + 1 * x) % f_i  = x^2 + 1
(x^2 + 1 * x) % f_i  = x^3 + x
(x^3 + x * x) % f_i  = x^2 + x + 1
(x^2 + x + 1 * x) % f_i  = x^3 + x^2 + x
(x^3 + x^2 + x * x) % f_i  = x^3 + x^2 + x + 1
(x^3 + x^2 + x + 1 * x) % f_i  = x^3 + x^2 + 1
(x^3 + x^2 + 1 * x) % f_i  = x^3 + 1
(x^3 + 1 * x) % f_i  = 1

p(x) = x^4 + x + 1
```

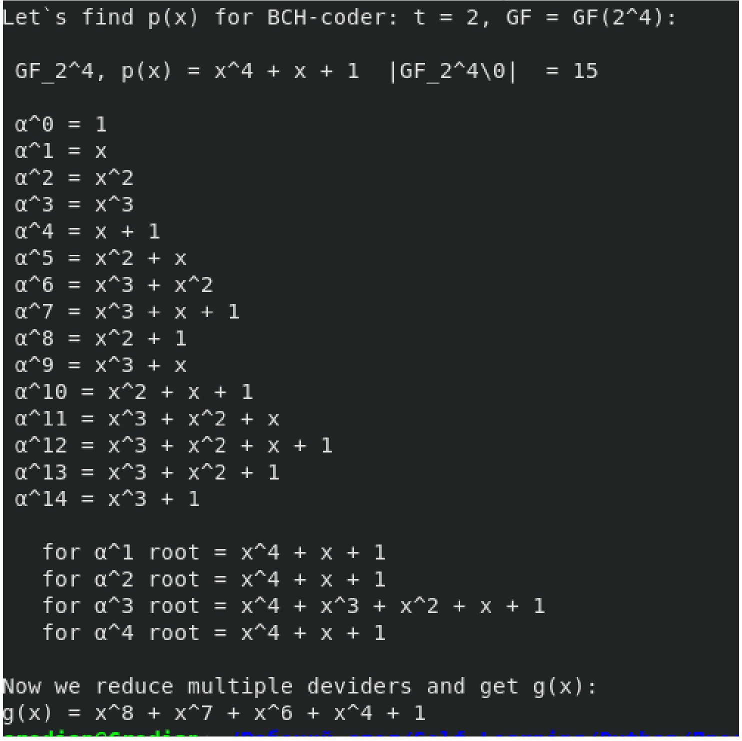
III . *Поиск генератора и величина информационного блока:*
- Из заданных выше параметров определяется величина информационного блока (сколько бит информационной последовательности помещается в один закодированный блок) – разность размера кодового блока и степени $g(x)$ - генератора.

Задача генератора – закодировать входную последовательность бит, воспринимаемую, как полином $v(x)$ в кодовый полином $c(x) = g(x) * v(x)$
- это умножение производится в $GF(2)$, а не в $GF(2)/p(x)$

По определению, $g(x)$ имеет корнями $\alpha^k,\ k \in \overline{1,\ \delta - 1}$ и при этом он минимальной степени(чтоб кодовый блок был как можно меньше)
другими словами: $g(\alpha) = 0$ при $\forall \alpha \in GF(2^m)$

Чтобы найти генератор – найдём минимальные полиномы, которые имеют корень α
Из всех найденных полиномов уберём повторяющиеся, оставшиеся – перемножим(т.е. посчитаем НОК для найденных полиномов)

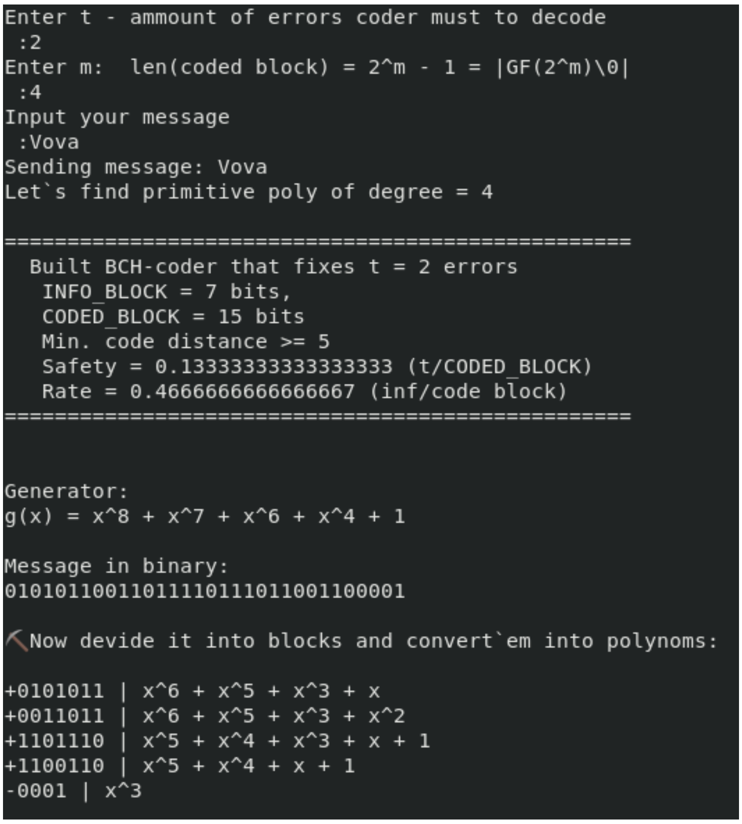
Ниже приведён пример поиска $g(x)$ для кода с параметрами $m = 4, t = 2$:



```
Let`s find p(x) for BCH-coder: t = 2, GF = GF(2^4):

 GF_2^4, p(x) = x^4 + x + 1  |GF_2^4\0|  = 15

 α^0 = 1
 α^1 = x
 α^2 = x^2
 α^3 = x^3
 α^4 = x + 1
 α^5 = x^2 + x
 α^6 = x^3 + x^2
 α^7 = x^3 + x + 1
 α^8 = x^2 + 1
 α^9 = x^3 + x
 α^10 = x^2 + x + 1
 α^11 = x^3 + x^2 + x
 α^12 = x^3 + x^2 + x + 1
 α^13 = x^3 + x^2 + 1
 α^14 = x^3 + 1

   for α^1 root = x^4 + x + 1
   for α^2 root = x^4 + x + 1
   for α^3 root = x^4 + x^3 + x^2 + x + 1
   for α^4 root = x^4 + x + 1

Now we reduce multiple deviders and get g(x):
g(x) = x^8 + x^7 + x^6 + x^4 + 1
```

Алгоритм кодирования, как было сказано выше: $c(x) = v(x)g(x)$

Для того, чтобы передать сообщение длиной(в битах) > информационный блок - “нарезаем” передаваемую последовательность битов(представляющих символы в нашем случае в кодировке ASCII) по размеру информационного блока, затем кодируем каждый информационный блок(воспринимая каждый бит, как коэффициент при степени x) – получаем кодовые блоки:



```
Enter t - ammount of errors coder must to decode
 :2
Enter m:  len(coded block) = 2^m - 1 = |GF(2^m)\0|
 :4
Input your message
 :Vova
Sending message: Vova
Let`s find primitive poly of degree = 4

==================================================
  Built BCH-coder that fixes t = 2 errors
   INFO_BLOCK = 7 bits,
   CODED_BLOCK = 15 bits
   Min. code distance >= 5
   Safety = 0.13333333333333333 (t/CODED_BLOCK)
   Rate = 0.4666666666666667 (inf/code block)
==================================================


Generator:
g(x) = x^8 + x^7 + x^6 + x^4 + 1

Message in binary:
01010110011011110111011001100001

⛏Now devide it into blocks and convert`em into polynoms:

+0101011 | x^6 + x^5 + x^3 + x
+0011011 | x^6 + x^5 + x^3 + x^2
+1101110 | x^5 + x^4 + x^3 + x + 1
+1100110 | x^5 + x^4 + x + 1
-0001 | x^3
```

Декодирование:

В результате прохождения кодовых блоков $c_i(x)$ по каналу передачи – в них возникает некоторое число ошибок и они принимают вид $c_i'(x) = c_i(x) + e_i(x)$, чтобы их исправить – воспользуемся декодированием, основанном на алгоритме Евклида.

I. *Декодирование на примере одного кодового блока $c'(x) = c(x) + e(x)$:*

В начале посчитаем синдромы $S_1, \ldots, S_{2t}$, где $S_i = c'(\alpha^i) = \overbrace{c(\alpha^i)}^{=0} + e(\alpha^i) = e(\alpha^i)$:

- если все синдромы нулевые – считаем, что ошибок не произошло: $c'(x) = c(x)$ и тогда $v(x) = c'(x)/g(x)$ – искомый информационный полином

- иначе в результате передачи произошла ошибка ⇒ есть ($S_i = e(\alpha^i)$) $\neq 0$.

Для определения позиций ошибок вводится полином – локатор ошибок $\sigma(y)$:

$$\sigma(y) = \left(1 + \alpha^{j_1} y\right) \ldots \left(1 + \alpha^{j_r} y\right)$$

где $\alpha^{j_i} -$ компонента полинома ошибок $e(x) = \alpha^{j_1} + \cdots + \alpha^{j_r}$

если мы сможем найти локатор(см. ниже), то его корни определят слагаемые из e(x):

$$\sigma(\alpha^k) = 0 \Leftrightarrow \left(1 + a^{j_i} \alpha^k\right) = 0 \Leftrightarrow a^{j_i} = a^{-k} = a^{n-k}$$

таким образом $a^{j_i} = a^{n-k}$ – слагаемое из e(x), где $n = |GF(2^m)\backslash\{0\}| = 2^m - 1$

Найдя все корни – определим все слагаемые e(x), зная e(x) – исправим все обнаруженные ошибки: $c'(x) + e(x) = c(x)$, тогда

$v(x) = (c'(x) + e(x))/g(x)$ – искомый информационный полином

II. *Поиск локатора ошибок $\sigma(y)$ по расширенному алгоритму Евклида:*

Вводятся понятия полинома синдромов $S(y)$ и полинома значений ошибок $w(y)$:

$$S(y) = \sum_1^{2t} S_i y^{i-1}\ , \quad w(y) = \sum_1^{r} \alpha^{j_i} \prod_{q=1,\ q\neq i}^{r} 1 + \alpha^{j_q}$$

путём преобразования сумм и переходом от равенств к сравнениям по модулю y^{2t} удаётся получить сравнение:

$w(y) \equiv S(y)\sigma(y)\ mod\ y^{2t}\ -$ ключевое уравнение

из определений имеем ограничения: $\deg \sigma(y) \leq t$ и $\deg w(y) \leq t-1$ **(1)**

Расширенный алгоритм Евклида находит коэффициенты c_a, c_b:

$$a(y)c_a + b(y)c_b = w(y) = \text{НОД}(a(y), b(y))$$

перейдём к сравнению по модулю $a(y)$:

$$a(y)c_a(y) + b(y)c_b(y) \equiv w(y)\ \ mod\ a(y) \Rightarrow b(y)c_b(y) \equiv w(y)\ \ mod\ a(y)$$

$$\Rightarrow w(y) \equiv b(y)c_b(y)\ \ mod\ a(y)\ \ (۞)$$

Легко видеть, что полученное выше сравнение (۞) легко обращается в ключевое уравнение, когда мы выполняем расширенный алгоритм Евклида для НОД(y^{2t}, S(y)):

$a(y) \to y^{2t},\ \ b(y) \to S(y)$

$w(y) \equiv b(y)c_b(y)\ mod\ a(y) \to\ w(y) \equiv S(y)c_b(y)\ mod\ y^{2t}$

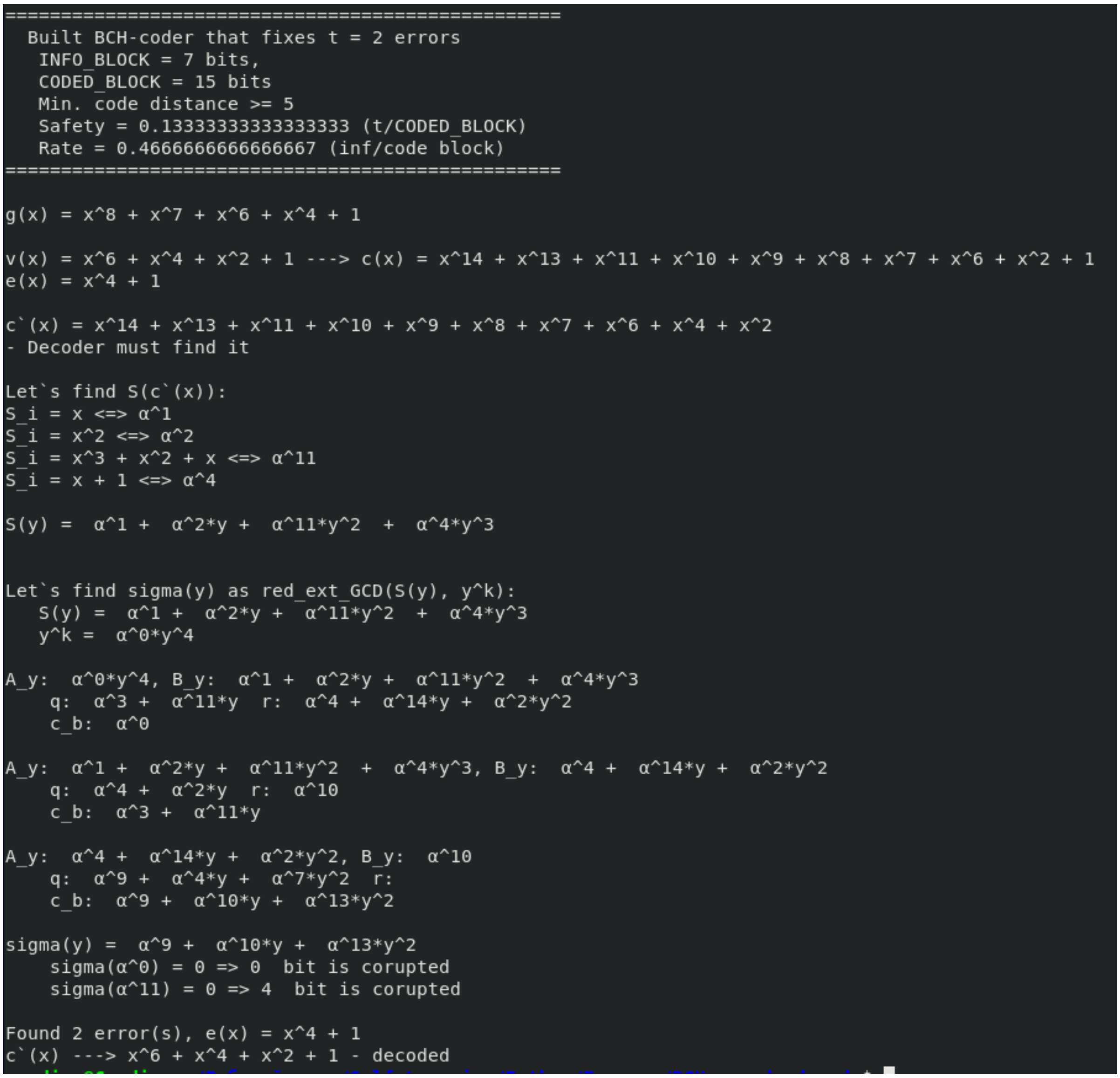
$w(y) \equiv S(y)\sigma(y)\ mod\ y^{2t}\ -$ ключевое уравнение

Получается $c_b(y) \equiv \sigma(y)$ – таким образом нашли локатор ошибок.

Однако до конца расширенный алгоритм Евклида выполнять не обязательно: сравнение (۞) истинно на каждом шаге алгоритма, при этом с каждым шагом растёт степень $c_b(y)$ и падает степень $w(y)$ – выполняем алгоритм до тех пор, пока не станут верны условия (1).

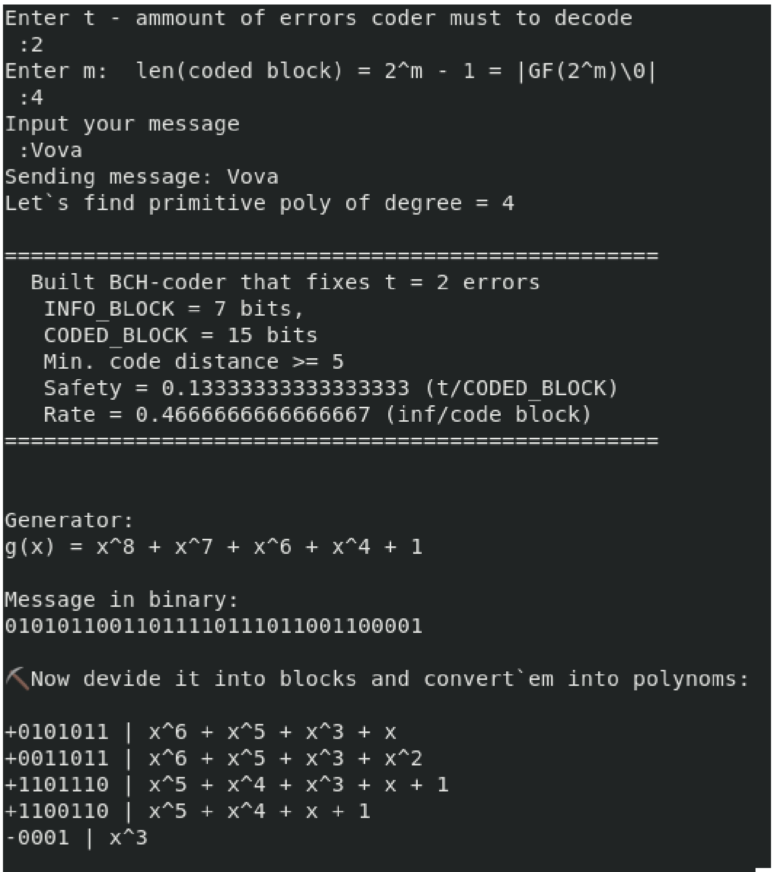
Полученная на последнем шаге(когда условия (1) стали верны) пара $(w(y), c_b(y))$ определит локатор, как и раньше: $\boldsymbol{c_b(y) \equiv \sigma(y)}$

Ниже приведён подробный пример кодирования и декодирования(в т.ч. поиск локатора sigma(y)) для информационного полинома $v(x) = x^7 + x^5 + x^3 + 1$:

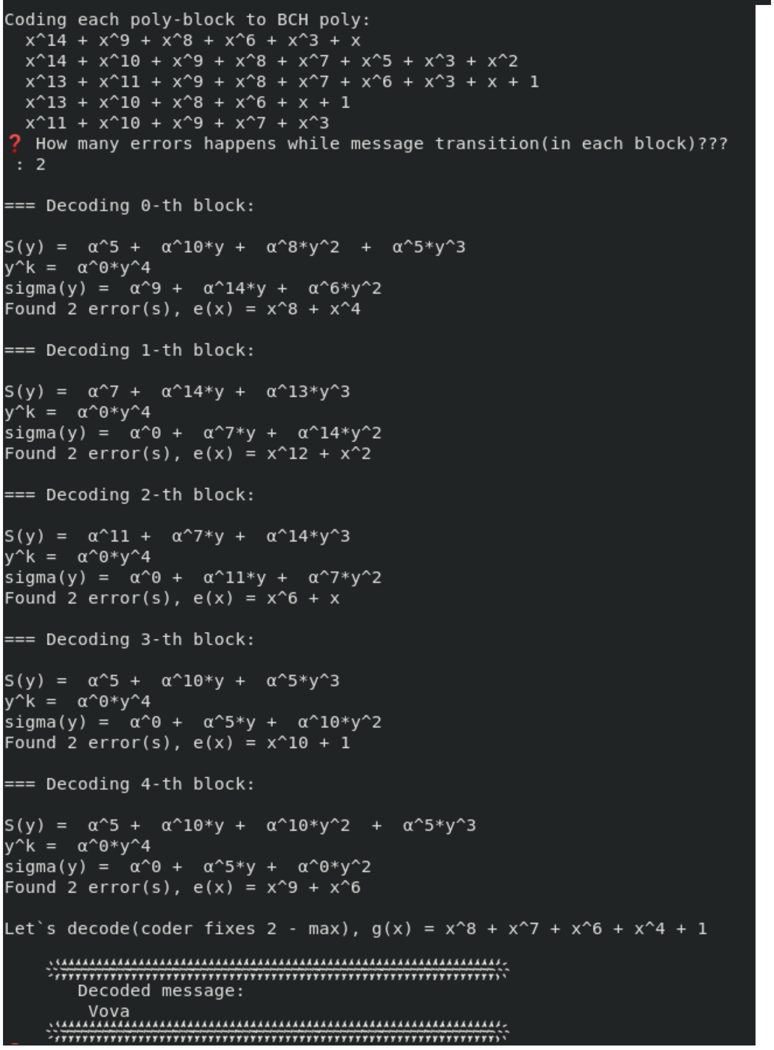


```
==================================================
  Built BCH-coder that fixes t = 2 errors
   INFO_BLOCK = 7 bits,
   CODED_BLOCK = 15 bits
   Min. code distance >= 5
   Safety = 0.13333333333333333 (t/CODED_BLOCK)
   Rate = 0.4666666666666667 (inf/code block)
==================================================

g(x) = x^8 + x^7 + x^6 + x^4 + 1

v(x) = x^6 + x^4 + x^2 + 1 ---> c(x) = x^14 + x^13 + x^11 + x^10 + x^9 + x^8 + x^7 + x^6 + x^2 + 1
e(x) = x^4 + 1

c`(x) = x^14 + x^13 + x^11 + x^10 + x^9 + x^8 + x^7 + x^6 + x^4 + x^2
- Decoder must find it

Let`s find S(c`(x)):
S_i = x <=> α^1
S_i = x^2 <=> α^2
S_i = x^3 + x^2 + x <=> α^11
S_i = x + 1 <=> α^4

S(y) =  α^1 +  α^2*y +  α^11*y^2  +  α^4*y^3


Let`s find sigma(y) as red_ext_GCD(S(y), y^k):
   S(y) =  α^1 +  α^2*y +  α^11*y^2  +  α^4*y^3
   y^k =  α^0*y^4

A_y:  α^0*y^4, B_y:  α^1 +  α^2*y +  α^11*y^2  +  α^4*y^3
    q:  α^3 +  α^11*y  r:  α^4 +  α^14*y +  α^2*y^2
    c_b:  α^0

A_y:  α^1 +  α^2*y +  α^11*y^2  +  α^4*y^3, B_y:  α^4 +  α^14*y +  α^2*y^2
    q:  α^4 +  α^2*y  r:  α^10
    c_b:  α^3 +  α^11*y

A_y:  α^4 +  α^14*y +  α^2*y^2, B_y:  α^10
    q:  α^9 +  α^4*y +  α^7*y^2  r:
    c_b:  α^9 +  α^10*y +  α^13*y^2

sigma(y) =  α^9 +  α^10*y +  α^13*y^2
    sigma(α^0) = 0 => 0  bit is corupted
    sigma(α^11) = 0 => 4  bit is corupted

Found 2 error(s), e(x) = x^4 + 1
c`(x) ---> x^6 + x^4 + x^2 + 1 - decoded
```

Далее рассмотрим пример кодирования и декодирования сообщения с разбиением на блоки (т.к. всё сообщение не помещается в один информационный блок):



```
Enter t - ammount of errors coder must to decode
 :2
Enter m:  len(coded block) = 2^m - 1 = |GF(2^m)\0|
 :4
Input your message
 :Vova
Sending message: Vova
Let`s find primitive poly of degree = 4

==================================================
  Built BCH-coder that fixes t = 2 errors
   INFO_BLOCK = 7 bits,
   CODED_BLOCK = 15 bits
   Min. code distance >= 5
   Safety = 0.13333333333333333 (t/CODED_BLOCK)
   Rate = 0.4666666666666667 (inf/code block)
==================================================


Generator:
g(x) = x^8 + x^7 + x^6 + x^4 + 1

Message in binary:
01010110011011110111011001100001

⛏Now devide it into blocks and convert`em into polynoms:

+0101011 | x^6 + x^5 + x^3 + x
+0011011 | x^6 + x^5 + x^3 + x^2
+1101110 | x^5 + x^4 + x^3 + x + 1
+1100110 | x^5 + x^4 + x + 1
-0001 | x^3
```

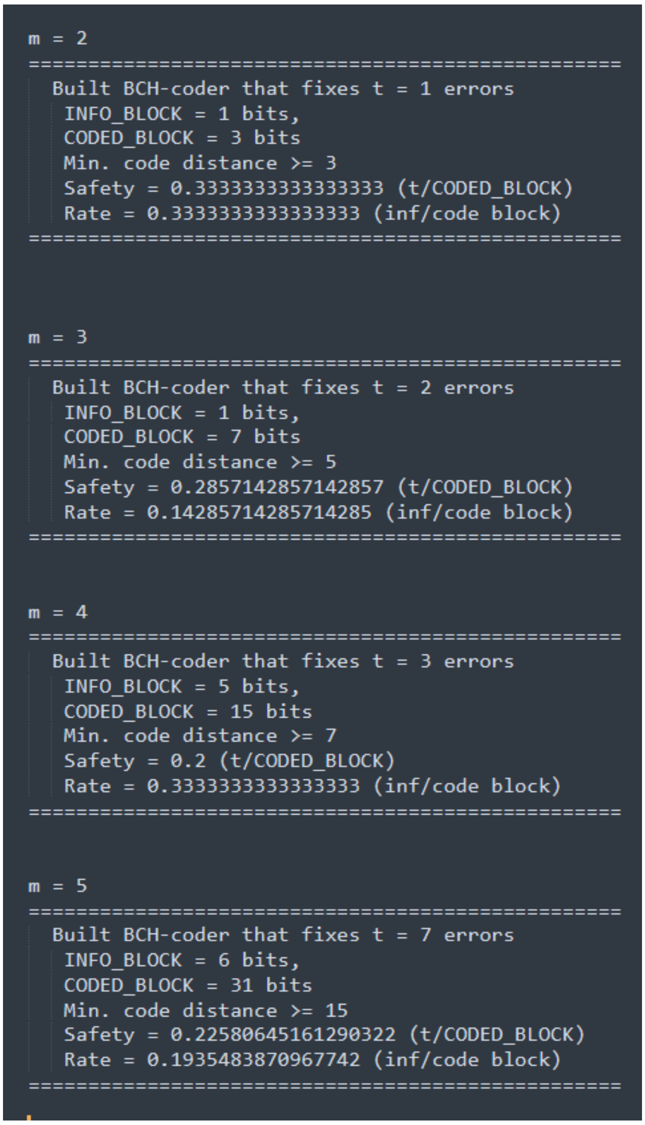


```
Coding each poly-block to BCH poly:
  x^14 + x^9 + x^8 + x^6 + x^3 + x
  x^14 + x^10 + x^9 + x^8 + x^7 + x^5 + x^3 + x^2
  x^13 + x^11 + x^9 + x^8 + x^7 + x^6 + x^3 + x + 1
  x^13 + x^10 + x^8 + x^6 + x + 1
  x^11 + x^10 + x^9 + x^7 + x^3
❓ How many errors happens while message transition(in each block)???
 : 2

=== Decoding 0-th block:

S(y) =  α^5 +  α^10*y +  α^8*y^2  +  α^5*y^3
y^k =  α^0*y^4
sigma(y) =  α^9 +  α^14*y +  α^6*y^2
Found 2 error(s), e(x) = x^8 + x^4

=== Decoding 1-th block:

S(y) =  α^7 +  α^14*y +  α^13*y^3
y^k =  α^0*y^4
sigma(y) =  α^0 +  α^7*y +  α^14*y^2
Found 2 error(s), e(x) = x^12 + x^2

=== Decoding 2-th block:

S(y) =  α^11 +  α^7*y +  α^14*y^3
y^k =  α^0*y^4
sigma(y) =  α^0 +  α^11*y +  α^7*y^2
Found 2 error(s), e(x) = x^6 + x

=== Decoding 3-th block:

S(y) =  α^5 +  α^10*y +  α^5*y^3
y^k =  α^0*y^4
sigma(y) =  α^0 +  α^5*y +  α^10*y^2
Found 2 error(s), e(x) = x^10 + 1

=== Decoding 4-th block:

S(y) =  α^5 +  α^10*y +  α^10*y^2  +  α^5*y^3
y^k =  α^0*y^4
sigma(y) =  α^0 +  α^5*y +  α^0*y^2
Found 2 error(s), e(x) = x^9 + x^6

Let`s decode(coder fixes 2 - max), g(x) = x^8 + x^7 + x^6 + x^4 + 1

       Decoded message:
        Vova
```

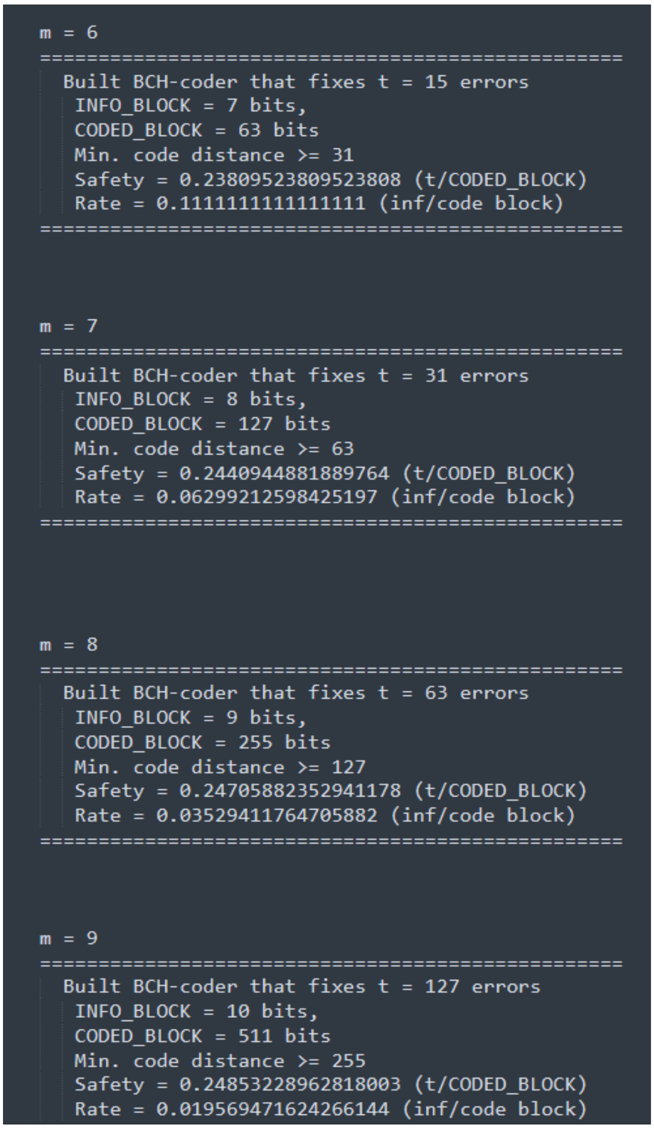
Оценка эффективности и поиск оптимальных параметров кода

Будем строить коды с максимально возможным числом исправляемых ошибок(ориентируясь не на Rate, а на Safety) для фиксированных m, m: $n = 2^m - 1 = |GF(2^m)/\{0\}|$, исключая вырожденные случаи, когда длина инф. блока = 1 биту(при $m > 3$)

Замечание: при фиксированном m, с ростом t – уменьшается “скорость” и наоборот



```
m = 2
==================================================
  Built BCH-coder that fixes t = 1 errors
   INFO_BLOCK = 1 bits,
   CODED_BLOCK = 3 bits
   Min. code distance >= 3
   Safety = 0.3333333333333333 (t/CODED_BLOCK)
   Rate = 0.3333333333333333 (inf/code block)
==================================================


m = 3
==================================================
  Built BCH-coder that fixes t = 2 errors
   INFO_BLOCK = 1 bits,
   CODED_BLOCK = 7 bits
   Min. code distance >= 5
   Safety = 0.2857142857142857 (t/CODED_BLOCK)
   Rate = 0.14285714285714285 (inf/code block)
==================================================


m = 4
==================================================
  Built BCH-coder that fixes t = 3 errors
   INFO_BLOCK = 5 bits,
   CODED_BLOCK = 15 bits
   Min. code distance >= 7
   Safety = 0.2 (t/CODED_BLOCK)
   Rate = 0.3333333333333333 (inf/code block)
==================================================


m = 5
==================================================
  Built BCH-coder that fixes t = 7 errors
   INFO_BLOCK = 6 bits,
   CODED_BLOCK = 31 bits
   Min. code distance >= 15
   Safety = 0.22580645161290322 (t/CODED_BLOCK)
   Rate = 0.1935483870967742 (inf/code block)
==================================================
```



```
m = 6
==================================================
  Built BCH-coder that fixes t = 15 errors
   INFO_BLOCK = 7 bits,
   CODED_BLOCK = 63 bits
   Min. code distance >= 31
   Safety = 0.23809523809523808 (t/CODED_BLOCK)
   Rate = 0.1111111111111111 (inf/code block)
==================================================


m = 7
==================================================
  Built BCH-coder that fixes t = 31 errors
   INFO_BLOCK = 8 bits,
   CODED_BLOCK = 127 bits
   Min. code distance >= 63
   Safety = 0.2440944881889764 (t/CODED_BLOCK)
   Rate = 0.06299212598425197 (inf/code block)
==================================================


m = 8
==================================================
  Built BCH-coder that fixes t = 63 errors
   INFO_BLOCK = 9 bits,
   CODED_BLOCK = 255 bits
   Min. code distance >= 127
   Safety = 0.24705882352941178 (t/CODED_BLOCK)
   Rate = 0.03529411764705882 (inf/code block)
==================================================


m = 9
==================================================
  Built BCH-coder that fixes t = 127 errors
   INFO_BLOCK = 10 bits,
   CODED_BLOCK = 511 bits
   Min. code distance >= 255
   Safety = 0.24853228962818003 (t/CODED_BLOCK)
   Rate = 0.019569471624266144 (inf/code block)
```

С увеличением m процент исправляемых ошибок в кодовом блоке хоть и растёт, но очень медленно(≤0,25).

Очевидно, код (m = 2, t = 1) – оптимален по обоим показателям(процент исправляемых ошибок в кодовом блоке и “скорость”), но этому есть и научное объяснение - данный код касается границы Синглтона:

$$k = n - d + 1 \iff 1 = 3 - 3 + 1$$

Замечание: граница Синглтона для удобства записана в логарифмированном виде, т.к. в коде используем основание q = 2.

Программная реализация

Файлы с исходным кодом:
poly_Z_2.py – реализация полиномов в $GF(2)$, поиск $p(x)$
GF_2_X.py – построение $GF(2^m)$ и определение операций в нём, поиск генератора.
BCH.py – кодирование строки(из кодировки ASCII в БЧХ), создание помех канала и декодирование сообщения.

Запуск:
python3 BCH.py

Исходные коды:
1. poly_Z_2.py:

```
'''
        List of crusual functions with poly_Z_2:

            0) there`s  +, -, *, /, % for polynoms
                also next() - itterator for polynoms

            1) Z2_prime_factors() - find all prime factors of poly (but dont count their
deg: ex: x^2 devides F(x) => x will be appended, due x^2 - isn`t prime) <=> canonical
factor without degrees
            2) get_primitive(t)   - gives primitive poly of deg = t that can permutate
GF(2^t)/{0} and returns GF(2^t)/{0}`s int koeffs for it`s polynoms
            3) on_val(g_x)        - returns value of f on g(x) <=> f(g(x))

'''




import copy              # for "phisical"(not by link) copping of objects



# polynoms as set of bits
class poly_Z_2:

    def __init__(self, *, koeffs = None, F = None, f_x:int = -1, str_koeffs = ""
):            # init poly by list of coefficents or F_x or ....

        if F != None:                           # by ecxisting polynom
            self.f_x = F.f_x
            self.deg = F.deg
```

```
        elif koeffs != None:                    # by list of koeffs
            self.f_x = 0
            self.deg = 1

            for i in range(0, len(koeffs)):
                self.f_x += koeffs[i] * 2**i

            self.update_deg()   # count degree

        elif f_x > -1:                          # by integer, which in binary represents
polynom
            self.f_x = f_x
            self.deg = 0        # just init value
            self.update_deg()   # count degree

        elif str_koeffs != "":                  # by String of zeroes and ones
            lst = [int(k) for k in str_koeffs]
            self.__init__(koeffs=lst)

        else:                                   # default constructor
            self.deg = 0
            self.f_x = 0



    def __add__(self, G):
        S = poly_Z_2()

        S.f_x = self.f_x ^ G.f_x
        S.update_deg()
        return S                #  "+" in GF_2 <=> XOR


    def update_deg(self):
        t_x = self.f_x
        t_deg = 0

        while t_x > 1:      # when t_x = 1 or 0  - degree is 0
            t_x   //= 2
            t_deg += 1

        self.deg = t_deg



    def __mul__(self, G):

        mul = poly_Z_2()
        for deg in range(0, self.deg + 1):

            if (self.f_x ^ 2**deg) < self.f_x:          # check if x^i excists (koeff by
x^i = 1 )
```

```
                mul.f_x ^= G.f_x << deg                 # XOR is same as "+" in GF2

        mul.update_deg()    # due GF2 some powers could kill themself( 2 = 0 in GF2)
        return mul





    def __mod__(self, G):

        # Подвинули битово влево на разницу степеней и поксорили
        # step of mod-deviding
        def sub_mod(Res : poly_Z_2, f_mod):
            diff = Res.deg - f_mod.deg         # find optimal degree of x

            q_i  = f_mod.f_x << diff           # get f_mod * x^diff <=> rotate left
"diff"-count bits
            Res.f_x ^= q_i                     # XOR is "-" in GF_2

            Res.update_deg()


        Res = poly_Z_2(F=self)                 # modulo-Remainder

        if self.deg < G.deg:
            return Res
        else:
            while Res.deg >= G.deg:
                sub_mod(Res, G)


        return Res


    def __truediv__(self, G):

        # Подвинули битово влево на разницу степеней и поксорили
        # step of mod-deviding
        def sub_div(Res : poly_Z_2, f_mod):
            diff = Res.deg - f_mod.deg         # find optimal degree of x
            q.f_x ^= 2**diff

            q_i  = f_mod.f_x << diff           # get f_mod * x^diff <=> rotate left
"diff"-count bits
            Res.f_x ^= q_i                     # XOR is "-" in GF_2
            Res.update_deg()

        Res = poly_Z_2(F=self)                 # modulo-Remainder
        q = poly_Z_2()

        while Res.deg >= G.deg:
```

```
            sub_div(Res, G)

        q.update_deg()
        return q








    # Phisical coppy(not by link) of poly.
    # !!! Use it except "="
    def coppy(self):
        return poly_Z_2(F=self)

    def get_deg(self):
        return self.deg

    def get_f_x(self):
        return self.f_x


    def __str__(self):

        if self.deg == 0:
            return f"{self.f_x % 2}"

        f_x_str = ""
        koefs = bin(self.f_x).lstrip("0b")                  # string with binary
representation of f_x(stores as number) without format-string: "0b"
        k_deg = self.deg


        for koef in koefs:
            if koef != "0":  f_x_str += f"x^{k_deg} + "
            k_deg -= 1

        # pretty output for 1-th and 0-th powers
        f_x_str = f_x_str.replace("x^1 ", "x ")
        f_x_str = f_x_str.replace("x^0 ", "1 ")

        return f_x_str.rstrip(" + ")


    def to_list(self):

        if self.deg == 0:
            return [self.f_x % 2]

        koefs = bin(self.f_x).lstrip("0b")                  # string with binary
representation of f_x(stores as number) without format-string: "0b"
```

```
        f_x_lst = [int(k) for k in koefs]

        return f_x_lst[ : :-1]                              # return inverse ordered
list


    # proto iterator for polynoms
    def next(self):
        F_i = copy.deepcopy(self)
        F_i.f_x += 1                              # itterate over polynoms:     x, x+1, x^2,
x^2 + 1, x^2 + x + 1, x^3, ....
        if F_i.f_x % 2**(F_i.deg + 1) == 0:   # check if degree has grown (phisicaly)
            F_i.deg += 1                          # renew degree

        return F_i

    # tells if polynom is prime
    def is_prime(self):
        F_i = poly_Z_2(koeffs=[0, 1])             # F_i = x

        while F_i.deg <= self.deg//2:
            if  (self % F_i).f_x == 0:
                return False

            F_i = F_i.next()
        return True


    def uniqals(lst):
        uniq = [lst[0]]
        for el in lst:
            if el.f_x not in [u.f_x for u in uniq]:   uniq.append(el)
        return uniq



    def sub_factor(F, F_i):

        prime_factors = []

        while F_i.deg <= F.deg - 1:

            if F_i.is_prime() and (F % F_i).f_x == 0:

                prime_factors.append( copy.deepcopy(F_i) )                 # adding
phisical coppy of F_i, bc object F_i will be modified later <=> F_i in list will change
too
                sub = poly_Z_2.sub_factor( (F / F_i), F_i.next())         # now
factorizing subfactor

                prime_factors.extend(sub)
```

```
                prime_factors = poly_Z_2.uniqals(prime_factors)            # returns only uniqual factors
                return prime_factors

            F_i = F_i.next()


        if len(prime_factors) == 0:
            prime_factors.append(F)
            return  prime_factors


    # returns only prime deviders of poly. ex: Z2_factorize(x^4 + x^2) = x, x^2 + 1 - not x^2, x^2 + 1, bc x^2 - not prime
    def Z2_prime_factors(self):
        F_i = poly_Z_2(koeffs=[0, 1])             # F_i = x - value to start reqursion

        return poly_Z_2.sub_factor(self, F_i)



    def pow(self, k: int):
        if k == 0:   return poly_Z_2(koeffs=[1])
        tmp = copy.deepcopy(self)
        for _ in range(1, k):
            tmp *= self
        return tmp

    def on_value(self, val):
        koefs = self.to_list()
        res = poly_Z_2()
        for i in range (0, len(koefs)):
            if koefs[i] != 0:   res += val.pow(i)

        return res

    # if deg = 13, 12 - 2 sec / 0.7 sec, if deg <= 11 - just a moment. Otherwise - too long
    def get_primitive(deg: int):
        #print(f"Let`s find primitive poly of degree = {deg}" )

        f_i = poly_Z_2(f_x=2**deg)
        alpha = poly_Z_2(f_x=2)       # x is always primitive
        while True:
            f_i = f_i.next()
            if f_i.is_prime():
                #print(f"Check GF/{f_i}, f_i = {f_i}:")
                test_GF = [1]                       # may be GF
                el = poly_Z_2(f_x=1)                # x^0 = 1
                while len(test_GF) < 2**deg - 1:    # while GF not fully filled
                    el = (el * alpha) % f_i
                    #print(f"({el} * {alpha}) % f_i  = {(el * alpha) % f_i}")
```

```
                    if el.f_x not in test_GF:
                        test_GF.append(el.f_x)
                    else:
                        #print(f" !!! element {(el * alpha) % f_i} is already in GF => go
to next prime poly:\n")
                        break                       # found same element => it`s not GF =>
take next poly
                else:
                    #print(f"\n-Finaly we build GF/{f_i}")
                    return f_i, test_GF             # if test_GF is GF(consists of uniqual
elements). ps test_GF

            if f_i.deg > deg:
                print(f"Something went wrong(BUG) and there is no primitive poly(but must
be) of deg = {deg}")
```

2. GF_2_X.py:

```

import copy
from poly_Z_2 import poly_Z_2

import random


'''
  Reduced pole: GF_2_X\{0} - there is no  0 element( useless in BCH and can`t be permute
from alpha by def)
   !!! Also GF_2_X\{0} - is multiplicative group of pole

    Some crusual functions:
        1) get_ord(), get_alpha_i() - get order of some poly in some GF
        2) mul(), mod(), dev()      - *, %, / - but cheaper - in GF
        3) get_g_x()                - get generator poly: g(alpha) = 0 for each apha in GF
'''



class GF2_x:

    def __init__(self, P: poly_Z_2, *, Table: list = None):
        self.alpha = poly_Z_2(koeffs = [0, 1])       # alpha = x is always primitive, so
let it be built-in
        self.p_x = P.coppy()

        # Table of int-represented polynoms - useful with poly_Z_2.get_primitive.  This
case - much cheapper
        if Table != None:
            self.table = [poly_Z_2(f_x=el) for el in Table]

        # counting each el of table
        else:
            self.table = [ poly_Z_2(koeffs = [1]) ]      # 0-th power of alpha = 1 is
always in table(GF_X)
```

```
            for i in range(1, 2**P.get_deg() - 1):
                self.table.append(  (self.table[i-1] * self.alpha) % P)  # counting each
alpha^i = alpha^i-1 * alpha mod p_x






    def get_alpha_i(self, i: int):
        return copy.deepcopy( self.table[i % len(self.table)] )

    def get_ord(self, v : poly_Z_2):
        i = 0
        for f in self.table:
            if f.f_x == v.f_x:  return i
            i += 1
        print(f"\n ==================== !!Alert!! ===================\n  {self}\n
!!Alert!! There is no ({v}) in GF(watch table above)")


    def get_table(self):
        return self.table


        # F^-1: F * F^-1 = 1
    def get_opposite(self, F:poly_Z_2):
        ord = self.get_ord(F)
        return copy.deepcopy( self.get_alpha_i(len(self.table) - ord) )


    # Cheap multiplying in GF_2^t using table:
    def mul(self, f: poly_Z_2, g: poly_Z_2):
        if f.f_x == 0 or g.f_x == 0:
            return poly_Z_2(f_x=0)

        i, j = self.get_ord(f),  self.get_ord(g)
        return copy.deepcopy( self.get_alpha_i( (i+j) % len(self.table)) )


    def pow(self, F: poly_Z_2, k: int):
        if k == 0:
            return poly_Z_2(koeffs=[1])

        ord = self.get_ord(F)
        return self.get_alpha_i( (ord * k) % len(self.table))


    def devide(self, F: poly_Z_2, G: poly_Z_2):
        if F.deg < G.deg:
            return poly_Z_2()
        return copy.deepcopy( self.get_alpha_i( (F.deg - G.deg) % len(self.table) ))
```

```


    def mod(self, F: poly_Z_2, G: poly_Z_2):
        return F - self.mul(G, self.devide(F, G))    # F - (F//G)*G . F//G - is


    def on_val(self, F, val):
        koefs = F.to_list()
        res = poly_Z_2()
        for i in range (0, len(koefs)):
            if koefs[i] != 0:   res += self.pow(val, i)

        return res % self.p_x


    # seek for minimal polyom g_x: g(alpha) == 0 <=> each alpha devides g(x)
    def get_g_x(self, t: int):
        devs = []

        P = poly_Z_2(f_x = 2**(2**self.p_x.deg) + 2)        # f(x) = x^(2^m) + x - this
poly always has deviders for g(x) in GF(2^m)
        P_devs = P.Z2_prime_factors()

        delta = 2*t + 1                                     # minimal code distanse

        for g_root in self.table[1:delta]:
            for poly in P_devs:
                if self.on_val(poly, g_root).get_f_x() == 0:
                    devs.append(poly)
                    #print(f"   for α^{self.get_ord(g_root)} root = {poly}")
                    break

        #print("\nNow we reduce multiple deviders and get g(x):")
        devs = poly_Z_2.uniqals(devs)                       # Оставляем только уникальные
делители <=> НОК

        g_x = poly_Z_2(f_x=1)
        for d in devs:      g_x *= d


        return g_x





    def __str__(self):
        deg = 0
        s = ""

        s +=  f"\n GF_2^{self.p_x.deg}, p(x) = {self.p_x}  |GF_2^{self.p_x.deg}\\0|  =
{len(self.table)}\n\n"
```

```
        for alpha in self.table:
            s += f" α^{deg} = {alpha} \n"
            deg += 1

        return s
```

3. BCH.py

```

from copy import deepcopy

from typing import List
from poly_Z_2 import poly_Z_2
from GF_2_X import GF2_x
import random



# Даты сдачи 27, 30, 31, (Г-401) 3,4 пары;  1, и не факт 2, 3


'''
 Polynoms(y) that has poly_Z_2 as koefficents by degrees of y. also I assume poly_Y`s 
koefs to bellong GF for easier multiplying(by primitive element powers)

'''
class poly_Y:
        def __init__(self, GF: GF2_x, *, koeffs: List[poly_Z_2] = [], koeffs_alpha: 
List[int] = []):

            self.GF = GF

            # koeffs must be without extra koeffs: ex:  x*y^2 => koeffs = [0, 0, x],  not 
[0, 0, x, 0]
            if koeffs != []:
                self.koeffs = deepcopy(koeffs)
                self.deg = len(koeffs) - 1

            elif koeffs_alpha != []:
                self.koeffs = [ GF.get_alpha_i(k) for k in koeffs_alpha ]        # fill 
koeffs of polynom(y) due GF table and given koeffs
                self.deg = len(koeffs_alpha) - 1

            # Creating emty poly_Y that can be filled appending elements
            else:
                self.koeffs = []    # nothing. even not a zero
                self.deg = -1       # when we`ll add an element to list of koeffs deg`ll 
be increased
```

```
        def eq(self, G):
            if G.deg != self.deg:
                return False
            for i in range(0, self.deg + 1):
                if self.koeffs[i].f_x != G.koeffs[i].f_x:
                    return False
            return True

        # updates degree and also deletes heading 0*y^t:  0*y^3 + x*y^2 + 1 ---> x*y^2 +
1  aka [1, x, 0] ---> [1, x]
        def update_deg(self):
            deg = len(self.koeffs) - 1
            if deg == -1: print("!!! Invalid polynom(not inited)"); self.deg = -
1                      # if poly is undefinded(no koeffs) => return invalid value

            for koef_x in self.koeffs[::-1]:     # try to find first y^deg with not zero
koefficent by it
                if koef_x.f_x != 0:
                    self.deg = deg
                    break
                self.koeffs.pop()               # delete empty(0) koefs by y of highest
degree. Ex: otherwise 0*y^3 + y^2 - unintuitive => delete 0*y^3
                deg -= 1
            else:
                self.deg = 0                        # all koeffs = 0 => self = 0
                self.koeffs = [poly_Z_2(f_x=0)] # all koeffs were poped => koefs = [], but
must be [0] - we`ve added here

        def __mul__(self, G):

            mul = poly_Y(GF=self.GF)
            for deg in range(0, self.deg + G.deg +1):
                mul.koeffs.append(poly_Z_2())                   # giving memory for
mul[self.deg, self.deg + G.deg]

            for deg in range(0, self.deg + 1):                  # +1 because ammount of
koeffs = degree of poly + 1
                if self.koeffs[deg].f_x != 0:
                    for G_deg in range(0, G.deg + 1):
                        if G.koeffs[G_deg].f_x != 0:
                            #print(f"    +++ to deg = {deg + G_deg} | {self.koeffs[deg]} *
{G.koeffs[G_deg]} = {self.GF.mul(self.koeffs[deg], G.koeffs[G_deg])}")
                            mul.koeffs[deg + G_deg] += self.GF.mul(self.koeffs[deg],
G.koeffs[G_deg])     # cheap multiplying from GF`s table

            mul.update_deg()
            return mul



        def __add__(self, G):
            sum = poly_Y(self.GF)
```

```
            if self.deg >= G.deg:
                for i in range(0, G.deg + 1):               # self and G poly may be of
different degree - so summ `em while they have same degrees
                    sum.koeffs.append((self.koeffs[i] + G.koeffs[i]) )
                for i in range(G.deg + 1, self.deg + 1):
                    sum.koeffs.append(self.koeffs[i])

            else:
                for i in range(0, self.deg +1):            # self and G poly may be of
different degree - so summ `em while they have same degrees
                    sum.koeffs.append((self.koeffs[i] + G.koeffs[i]) )
                for i in range(self.deg +1, G.deg +1):
                    sum.koeffs.append(G.koeffs[i])

            sum.update_deg()                            # modulo could decreaze power from
self.deg + G.deg to someth lower
            return sum


        # Итерратор по poly_Y
        def next(self):
            F_y = deepcopy(self)

            for i in range(0, len(F_y.koeffs)):
                if F_y.koeffs[i].f_x != 2**(self.GF.p_x.deg) - 1:     # check if koef
isn`t max (in GF(2^t)) and can be increased
                    F_y.koeffs[i] = F_y.koeffs[i].next()
                    for j in range(0, i):
                        F_y.koeffs[j] = poly_Z_2()
                    return F_y

            else:                                   # if all koeffs are max
                for i in range(0, len(F_y.koeffs)):
                    F_y.koeffs[i] = poly_Z_2()           # zeroing all the koeffs
                F_y.koeffs.append(poly_Z_2(f_x = 1))     # append new(and minimal) koeff
to the end
                F_y.deg += 1
                return F_y

        # return y^k
        def y_k(self, k: int):
            y_k = poly_Y(GF=self.GF)
            for deg in range(0, k):
                y_k.koeffs.append(poly_Z_2())   # append k - count zero koeffs

            y_k.koeffs.append(poly_Z_2(f_x=1))  # one more element that`s = 1
            y_k.deg = k
            return y_k
```

```
        # For poly_Y: return modulo(self % G) and devider(self/G)
        def mod_dev(self, G):

            def sub(res:poly_Y, div: poly_Y):
                k = res.deg - div.deg
                dev = self.y_k(k)                                                   # y^k
                dev.koeffs[-1] = self.GF.mul(res.koeffs[-
1],  self.GF.get_opposite(div.koeffs[-1]) )   # biggest koef by res and div_i must be 
equal to continue deviding => dev = res[-1]/div[-1]. ps /div[-1] <=> * div[-1]^-1

                div_i = div * dev                                                   # and 
here dev[-1] * div[-1] <=> div_i[-1] = res[-1]

                return res + div_i, dev             # res - div  is same as  res + div in 
GF(2^t)


            res = deepcopy(self)
            dev = poly_Y(self.GF, koeffs=[poly_Z_2(f_x = 0)])   # 0 * y^0 - 0 as F(y)

            while res.deg > G.deg:
                res, dev_i = sub(res, G)
                dev += dev_i

            # Ex: res = 0, dev_i = alpha^9 - this might cycle forewer above(due no changes 
and condition remains true) - so I separated this case
            if res.deg == G.deg:
                res, dev_i = sub(res, G)
                dev += dev_i

            return res, dev


        def on_val(self, F: poly_Z_2):
            Val = poly_Z_2()
            deg = 0
            for f in self.koeffs:
                if f.f_x != 0:
                    Val += self.GF.mul(f, self.GF.pow(F, deg))
                deg += 1
            return Val



        def __str__(self):

            f_y_str = ""
            k_deg = 0
            for koef in self.koeffs:
                if koef.f_x != 0:  f_y_str += f" α^{self.GF.get_ord(koef)}*y^{k_deg}  + 
"     # ({koef}) alph^{self.GF.get_ord(koef)}
                k_deg += 1
```

```
            # pretty output for 1-th and 0-th powers
            f_y_str = f_y_str.replace("y^1 ", "y")
            f_y_str = f_y_str.replace("*y^0 ", "")

            return f_y_str.rstrip(" + ")


# Some exceptions:
class Missfit_t_and_m(Exception):
    def __init__(self, t, m):
        # переопределяется конструктор встроенного класса `Exception()`
        sep = " \n" + "!"*40 + "\n"
        super().__init__(f"{sep} 2^{m} - 1 < minimal code distance({2*t + 1}) - u should increase m too fix t={t} errors{sep}")

class Too_small_m(Exception):
    def __init__(self, m):
        sep = "\n" + "!"*40 + "\n"
        super().__init__(f"\n m = 1 is too small{sep}")

class Error_too_big_message(Exception):
    def __init__(self, Message, INFO_BLOCK):
        sep = "\n" + "!"*40 + "\n"
        super().__init__(f"\n{sep} len(Message) in bits = {len(Message)}*8 = {len(Message)*8},\nwhile INFO_BLOCK size = {INFO_BLOCK} bits {sep}")

class BAD_Infoblock(Exception):
    def __init__(self, INFO_BLOCK):
        sep = "\n" + "!"*40 + "\n"
        super().__init__(f"\n{sep} INFO_BLOCK is too small = {INFO_BLOCK} bits {sep} for chosen m. Please, increase m")



class BCH:

    # t - ammount of errors fixing,  m - defines size of block = 2**m -1 = |GF(2**m)/{0}|
    def __init__(self, t: int, m: int):
        self.t = t

        if m < 2:
            raise Too_small_m(m)

        # minimal code distance >= 2t + 1  --->  n=2^m-1  >= 2*t + 1
        if 2**m - 1 < 2*t + 1:
            raise Missfit_t_and_m(t,m)

        # if deg(p(x))=m <= 10 works fast
        p, GF_tbl = poly_Z_2.get_primitive(m)       # find poly that can permutate all GF(2^m)/{0} elements.
        self.GF_x = GF2_x(p, Table=GF_tbl)          # build GF(2^m)
```

```
        # g`s roots = GF_x.table[0:delta] : alpha^0 ... alpha^(2t+1) - roots of g(x). delta = 2t+1 - minimal code distance
        self.g_x = self.GF_x.get_g_x(t)             # g(x) - generator. p(x) - generator. because it`s primitive poly(permutes every != 0 poly of deg < m)

        self.CODED_BLOCK = 2**m - 1
        self.INFO_BLOCK = self.CODED_BLOCK - self.g_x.deg
        if self.INFO_BLOCK == 1 and m > 3:
            raise BAD_Infoblock(self.INFO_BLOCK)

        sep = "\n" + "=" * 50 + "\n"
        print(f"{sep}  Built BCH-coder that fixes t = {t} errors \n   INFO_BLOCK = {self.INFO_BLOCK} bits, \n   CODED_BLOCK = {self.CODED_BLOCK} bits\n   Min. code distance >= {2*t + 1}\n   Safety = {t/self.CODED_BLOCK} (t/CODED_BLOCK)\n   Rate = {self.INFO_BLOCK/self.CODED_BLOCK} (inf/code block) {sep}")



    # some chanel distortion
    def distortion(self, k: int):

        dirty = poly_Z_2()

        for _ in range(0, k):
            i = random.randint(0, len( bin(self.CODED_BLOCK).lstrip("0b")) )
            dirty.f_x += 2**i

        dirty.update_deg()

        return dirty


    # messege --> poly-blocks
    def str_encode(self, message: str):
        mess = bytes(message, encoding = "ASCII")
        enc = ""

        # bytes --> binary string
        for ch in mess:
            #print(f" {ch} {bin(ch)}")
            enc += '{:0>8b}'.format(ch)         # Adding 8 charectered string with heading(>) zeroes(0) for binary(b) representation of ch.  see "format_spec" - syntax in format()

        print(f"Message in binary: \n{enc}\n\n{chr(9935)} Now devide it into blocks and convert`em into polynoms:\n")

        # slicing binary string into blocks of INFOBLOCK size
        blocks_x = []
        i = 0
```

```
        if len(enc) < self.INFO_BLOCK:
            blocks_x.append( poly_Z_2(str_koeffs=enc) )
            return  blocks_x


        while i < len(enc):
            if len(enc) - i - self.INFO_BLOCK >= 0:
                print(f"+{enc[i:i+self.INFO_BLOCK]} |
{poly_Z_2(str_koeffs=enc[i:i+self.INFO_BLOCK])} ")
                blocks_x.append( poly_Z_2(str_koeffs=enc[i:i+self.INFO_BLOCK]) )
            else:
    # in case last block can`t be of size INFO_BLOCK
                print(f"-{enc[i:i + len(enc) - self.INFO_BLOCK]} |
{poly_Z_2(str_koeffs=enc[i: i + len(enc) - self.INFO_BLOCK])} ")
                blocks_x.append( poly_Z_2(str_koeffs=enc[i:i + len(enc) -
self.INFO_BLOCK]))     # len(enc) - self.INFO_BLOCK - ammount of remained bits
            i += self.INFO_BLOCK

        return blocks_x


    # poly-blocks ---> messege
    def str_decode(self, blocks_x:list):

        bin_repr = ""
        for F in blocks_x:
            bin_repr += bin(F.f_x).lstrip("0b")[::-1]                                   #
integers in polynoms stores in reversed order
            bin_repr += "0" * (self.INFO_BLOCK - len(bin(F.f_x).lstrip("0b")[::-1]))    #
padding to length of info block each block(ex: 2 = 10 ---> 0010 if inf_block = 4)

        bin_repr = bin_repr + "0" * (len(bin_repr) % 8)   # padding to byte sized str. ps
byte = 8 bits

        dec_str = ""
        i = 0
        #print("V:  " + bin(ord("V")))
        while i < len(bin_repr):
            #print("   " + bin_repr[i: i+8])
            dec_str += chr( int("0b" + bin_repr[i: i+8], base=2))
            i += 8


        return dec_str



    # syndrom-polynom(y): each i-th koeff = value of recieved code-vector(v`) on i-th
power of alpha from GF_X
    def get_S_y(self, recived: poly_Z_2):       # recived - v`(x) - code-message with some
distortion from chanel
```

```
        S_y = poly_Y(self.GF_x)
        koefs = recived.to_list()               # getting representation of polynom as list of it`s koeffs

        # give memory for each S(y)`s koeff
        for deg in range(0, 2 * self.t):
            S_y.koeffs.append(poly_Z_2())

        deg = 0                                 # degree of y
        for k in koefs:
            if k != 0:
                for i in range(0, 2 * self.t):
                    S_y.koeffs[i] += self.GF_x.pow(self.GF_x.get_alpha_i(deg), i+1)   # Updating all koeffs of S(y): S(y)_i += (alpha^deg)^(i+1). ps i+1 bc i migh be 0
            deg +=1


        S_y.update_deg()

        '''
        print("Let`s find S(c`(x)):")
        for S_i in S_y.koeffs:
            if S_i.f_x != 0:
                print(f"S_i = {S_i} <=> α^{self.GF_x.get_ord(S_i)}")
        print(f"\nS(y) = {S_y}\n")
        '''

        return S_y                              # S(y) = alpha^k_1 + alpha^k_2 * y^1 + ... + alpha^k_2t(x) * y^2t-1  - this poly(y) stores as list of koeffs(has type f(x))



    # apaptation(reducing some steps) of extended Euclid`s algorithm for BCH`s purposes(finding sigma(y))
    # deg a(x) >= deg b(x)
    def reduced_ext_GCD(self, a: poly_Y, b: poly_Y):

        A_y, B_y  = deepcopy(a), deepcopy(b)                    # create physical coppies of givven polynoms not to corrupt `em`
        q, r      = poly_Y(self.GF_x), poly_Y(self.GF_x)
        c_b, c_b_ = poly_Y(self.GF_x, koeffs=[poly_Z_2(f_x=0)]), poly_Y(self.GF_x, koeffs=[poly_Z_2(f_x=1)])      # 0 and 1 as polynoms

        zero_Y = poly_Y(self.GF_x, koeffs=[poly_Z_2()])                 # 0  in poly_y type


        while (not B_y.eq(zero_Y)) and A_y.deg >= self.t:
            print(f"A_y: {A_y}, B_y: {B_y}")
            r, q = A_y.mod_dev(B_y)
            print(f"    q: {q}  r: {r}")
```

```
            A_y,B_y = B_y, r
            c_b, c_b_ = c_b_, c_b + q * c_b_

            print(f"    c_b: {c_b}\n")

        return c_b      # almost sigma(y). koeff by sigma by def is 1, so we can multiply
on inverted koeff[0] all the sigma(y), but it`s roots`ll remain same, so it`s useless

    # messege as poly-blocks ---> BCH-encoded poly-blocks
    def encode(self, blocks_x: list):
        print("\nCoding each poly-block to BCH poly:")
        enc_blocks = []
        for F in blocks_x:
            print(f"  {F*self.g_x}")
            enc_blocks.append(F*self.g_x)

        return enc_blocks

    def is_ok(self, koeffs, val):
        for el in koeffs:
            if el.f_x != val:
                return False
        return True


    # BCH-decoder - for single block. Multiple blocks - use Terminal
    # BCH-encoded poly ---> messege poly
    def decode(self, v_: poly_Z_2):

        e = poly_Z_2()                  # polynom of errors
        S_y = self.get_S_y(v_)          # syndrom(y)

        if self.is_ok(S_y.koeffs, 0):   # if no errors
            print("There`s no errors")
            return deepcopy(v_/self.g_x)


        print(f"\nLet`s find sigma(y) as red_ext_GCD(S(y), y^k):\n   S(y) = {S_y}\n   y^k
= {poly_Y(self.GF_x).y_k(2 * self.t)}\n")

        sigma_y = self.reduced_ext_GCD(poly_Y(self.GF_x).y_k(2 * self.t), S_y)  #
red_GCD(y^2y, S_y) => locator

        print(f"sigma(y) = {sigma_y}")

        #print(f"\nS(y) = {S_y}\ny^k = {poly_Y(self.GF_x).y_k(2 * self.t)}\nsigma(y) =
{sigma_y}")

        # locator aka sigma(y) by def: sigma(y) = (1 + a^j1 * y)(1 + a^j2 * y)...(1 + a^jr
* y), where j - is position of error
        err_counter = 0
        for alpha in self.GF_x.table:
```

```
            if sigma_y.on_val(alpha).f_x == 0:                      # Locator(alpha) = 0
<=> alpha is root =>  alpha^-1 is in e(x)

                err_counter += 1                                    # new error find
                opos = self.GF_x.get_opposite(alpha)                # alpha^-1 = alpha^j
=> j = get_ord(alpha^-1)
                e += poly_Z_2(f_x = 2 ** self.GF_x.get_ord(opos))   # v += x^j
                print(f"    sigma(α^{self.GF_x.get_ord(alpha)}) = 0 => {self.GF_x.get_ord(
self.GF_x.get_opposite(alpha) )}  bit is corupted")


        if err_counter > self.t:
            print(f"\nThere`s more than {self.t}(max) errors \\_(0_0)_/ ")
            return 0
        else:
            print(f"\nFound {err_counter} error(s), e(x) = {e}")
            return (v_ + e) /self.g_x                              # reduce e from v_ .ps
+ and - same GF(2)


def Terminal(message: str, t_errors: int, m: int):
    print(f"Sending message: {message}")

    Coder = BCH(t_errors, m)
    print(f"\nGenerator:\ng(x) = {Coder.g_x}\n")
    v_blocks = Coder.str_encode(message)
    enc_blocks = Coder.encode(v_blocks)

    k = int(input(f"{chr(10067)} How many errors happens while message transition(in each
block)???\n : "))
    lst_e_x = [Coder.distortion(k) for block in
enc_blocks]                                # forming distortion for each block

    dec_blocks = []
    for i in range(0, len(enc_blocks)):
        print(f"\n=== Decoding {i}-th block:")
        dec_blocks.append( Coder.decode(enc_blocks[i] + lst_e_x[i]) )

    print(f"\nLet`s decode(coder fixes {t_errors} - max), g(x) = {Coder.g_x}")


    dec_str = Coder.str_decode(dec_blocks)

    magic = "      " + chr(1161)*30
    print(f"\n{magic*7}\n       Decoded message:\n        {dec_str}\n{magic*7}")


    ans = input(f"{chr(10067)} Wanna see used GF?[y]\n:")
    if ans == "y":
        print(Coder.GF_x)
```

```
if __name__ == "__main__":


    GF2_5 = GF2_x(poly_Z_2(koeffs=[1, 0, 1, 0, 0, 1]))       # x^5 + x^2 + 1 - primitive poly
    GF2_4 = GF2_x(poly_Z_2(koeffs=[1, 1, 0, 0, 1]))       # x^4 + x + 1 - primitive poly


    t = int(input("Enter t - ammount of errors coder must to decode\n :"))
    m = int(input("Enter m:  len(coded block) = 2^m - 1 = |GF(2^m)\\0|\n :"))
    message = input("Input your message\n :")
    Terminal(message, t, m)
```

Заключение

В данной работе получилось реализовать помехоустойчивый код БЧХ, убедится на практике в его работоспособности и исследовать его(поиск оптимальных параметров и выявление зависимостей характеристик кода от этих параметров).

Список литературы

1. Е.Г. Власов “Конечные поля в телекоммуникационных приложениях. Теория и применение FEC, CRC, M-последовательностей”
2. Прокис Дж. “Цифровая связь”