

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение высшего образования
«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
АЭРОКОСМИЧЕСКОГО ПРИБОРОСТРОЕНИЯ»

КАФЕДРА 33

ЛАБОРАТОРНАЯ РАБОТА
ЗАЩИЩЕНА С ОЦЕНКОЙ

РУКОВОДИТЕЛЬ

канд. техн. наук

должность, уч. степень, звание

В.В. Давыдов

подпись, дата

инициалы, фамилия

ЛАБОРАТОРНАЯ РАБОТА №2

«МОДЕЛИРОВАНИЕ СИСТЕМЫ ПОМЕХОУСТОЙЧИВОЙ СВЯЗИ.
ЧАСТЬ ВТОРАЯ»

по дисциплине: ТЕОРИЯ КОДИРОВАНИЯ

РАБОТУ ВЫПОЛНИЛ

СТУДЕНТ ГР. №

3333

подпись, дата

инициалы, фамилия

Санкт-Петербург 2025

Цель работы

Разработать программный модуль на выбранном языке программирования для построения и применения линейного блокового двоичного кода, способного обнаруживать и исправлять одиночные ошибки в передаваемых данных.

Задание к лабораторной работе

Вариант 1. По заданным параметрам $n \leq 15$ строить двоичный код БЧХ, выводить на экран порождающую и проверочную матрицы, а также порождающий и проверочный многочлен. Кодировать сообщение, введённое с клавиатуры. Декодировать сообщение с помощью декодера – Питерсона-Горенстейна-Цирлера.

Теоретические сведения

Линейный код:

Пусть q – степень простого числа, тогда:

$$C \leq F_q^n - \text{линейный код}$$

Если C – подпространство пространства F_q ;

F_q^n – пространство всех векторов длины n .

Циклический код:

Циклический код длины n с k информационными символами называется такой линейный n,k – код, у которого циклический сдвиг любого кодового слова так же является кодовым словом.

Код БЧХ:

Циклический код n над GF_q называется кодом БЧХ с конструктивным расстоянием d_0 , если для некоторого целого числа $m_0 > 0$ его порождающий мн-ен равен:

$$g(x) = \text{НОК}\{M_{(x)}^{(m_0)}, M_{(x)}^{(m_0+1)}, \dots, M_{(x)}^{(m_0+d_0-2)}\}$$

, где $M_{(x)}^{(i)}$ – мин. мн-ен

Код БЧХ в узком смысле:

Если $m_0 = 1$, то коды называются кодами БЧХ в узком смысле.

Примитивный/не примитивный БЧХ код:

Если $n = q^m - 1$, то коды называются примитивными кодами БЧХ;

Если $n \neq q^m - 1$, то коды называются не примитивными кодами БЧХ;

Граница БЧХ:

Пусть $g(x)$ – порождающий мн-ен циклического кода Γ , тогда его корни лежат в $GF(q)$, запишем их в виде степеней примитивных эл-тов:

$$\alpha^{i1}, \alpha^{i2}, \dots, \alpha^{ir}$$

Самая длинная из таких последовательностей: $m_0, m_0+1, \dots, m_0 + d_0 - 2$, тогда:

$$d_0 \Rightarrow d$$

Кодирование произвольного слова:

$$C(x) = m(x) \cdot g(x) \bmod (x^n - 1)$$

Декодирование методом ПГЦ:

Алгоритм:

- 1)Находим все S ;
- 2)Строим матрицу для S ;
- 3)Находим локаторы;
- 4)Находим корни $A(x)$;
- 5)Находим значения ошибок (e).

Ход работы

1 Поле GF(2^m): строим exp/log таблицы для быстрых умножений

Функция `gf_tables(m)` (таблица 1) создаёт таблицы `exp` и `log` для быстрого выполнения арифметических операций (умножения, деления, возведения в степень) в поле $GF(2^m)$.

Таблица 1 – `gf_tables(m)`

```
def gf_tables(m):
    """
    Строим таблицы:
    exp[i] =  $\alpha^i$ 
    log[a] = i, если  $a = \alpha^i$ 
    где  $\alpha$  — примитивный элемент поля  $GF(2^m)$ 
     $n = 2^m - 1$  — порядок мультиплексивной группы
    """
    n = 2**m - 1
    prim = PRIM[m]

    exp = [0] * (2 * n)
    log = [0] * 2**m

    exp[0] = 1
    log[0] = -1
    log[1] = 0

    # Генерируем  $\alpha^i$  последовательно
    for i in range(1, n):
        x = exp[i - 1] << 1
        # Если "вылез" старший бит, делаем редукцию по примитивному полиному
        if x & (1 << m):
            x ^= prim
        exp[i] = x & ((1 << m) - 1)
        log[exp[i]] = i

    # Удобно продублировать exp, чтобы не писать модуль каждый раз
```

```

for i in range(n, 2 * n):
    exp[i] = exp[i - n]

return exp, log, n

```

Функция `gf_mul(a, b, exp, log, n)` (таблица 2) выполняет умножение двух элементов a и b в поле $GF(2^m)$, используя таблицы `exp` и `log`.

Таблица 2 – `gf_mul(a, b, exp, log, n)`

```

def gf_mul(a, b, exp, log, n):
    """Умножение в GF(2^m) через таблицы log/exp."""

    if a == 0 or b == 0:
        return 0

    return exp[(log[a] + log[b]) % n]

```

Функция `gf_inv(a, exp, log, n)` находит мультипликативно обратный элемент a^{-1} в поле $GF(2^m)$, используя таблицы `exp` и `log`.

Таблица 3 – `gf_inv(a, exp, log, n)`

```

def gf_inv(a, exp, log, n):
    """

Обратный элемент в GF(2^m):  $a^{-1} = a^{n - \log(a)} = a^{-\log(a)}$ .

Используем  $\exp[-\log[a]]$ , что эквивалентно  $\exp[n - \log[a]]$  благодаря периодичности.

"""

    if a == 0:
        raise ZeroDivisionError("Обратный элемент для 0 не существует")

    return exp[-log[a]]

```

2 Полиномы над GF(2)

Функция trim(p) (таблица 4) удаляет ведущие (в конце списка) нулевые коэффициенты из списка, представляющего полином.

Таблица 4 – trim(p)

```
def trim(p):
    """Убираем ведущие нули (с конца списка)."""
    while len(p) > 1 and p[-1] == 0:
        p.pop()
    return p
```

Функция pmul2(p, q) (таблица 5) выполняет умножение двух полиномов p и q над полем GF(2) (коэффициенты 0 или 1), где сложение — это XOR.

Таблица 5 - pmul2(p, q)

```
def pmul2(p, q):
    """Умножение полиномов над GF(2). Сложение = XOR."""
    r = [0] * (len(p) + len(q) - 1)
    for i, a in enumerate(p):
        if a:
            for j, b in enumerate(q):
                if b:
                    r[i + j] ^= 1
    return trim(r)
```

Функция pdiv2(dividend, divisor) (таблица 6) выполняет деление полинома dividend на divisor над полем GF(2), возвращая частное и остаток.

Таблица 6 - pdiv2(dividend, divisor)

```
def pdiv2(dividend, divisor):
    """
    Деление полиномов над GF(2):
    dividend = divisor * quotient + remainder
    """
    dd = trim(dividend[:])
    dv = trim(divisor[:])
    q = [0] * max(0, len(dd) - len(dv) + 1)
```

```

while len(dd) >= len(dv) and dd != [0]:
    sh = len(dd) - len(dv)
    q[sh] = 1
    # dd = dd + (dv * x^sh) (в GF(2) '+' это XOR)
    for i, c in enumerate(dv):
        if c:
            dd[i + sh] ^= 1
    dd = trim(dd)

return trim(q), trim(dd)

```

Функция pstr(p) (таблица 7) преобразует список коэффициентов полинома p в читаемую строку (например, [1, 1, 0, 1] -> "x^3 + x^2 + 1").

Таблица 7 - pstr(p)

```

def pstr(p):
    """Красивый вывод полинома над GF(2)."""
    t = []
    for i, c in enumerate(p):
        if c:
            t.append("1" if i == 0 else ("x" if i == 1 else f"x^{i}"))
    return "0" if not t else "+" .join(reversed(t))

```

3 Минимальные многочлены: работаем во временных коэффициентах

GF(2^m)

Функция pmul_gf(p, q, exp, log, n) (таблица 8) выполняет умножение двух полиномов p и q, коэффициенты которых принадлежат полю GF(2^m), используя gf_mul для умножения коэффициентов.

Таблица 8 - pmul_gf(p, q, exp, log, n)

```
def pmul_gf(p, q, exp, log, n):
```

```
    """Умножение полиномов над GF( $2^m$ ) (коэффициенты - элементы поля)."""
```

```
    r = [0] * (len(p) + len(q) - 1)
```

```
    for i, a in enumerate(p):
```

```
        if a:
```

```
            for j, b in enumerate(q):
```

```
                if b:
```

```
                    r[i + j] ^= gf_mul(a, b, exp, log, n) # сложение в поле = XOR
```

```
    return r
```

Функция coset(i, n) (таблица 9) находит циклотомический класс элемента i по модулю n, то есть набор значений $\{i, 2i, 4i, \dots\}$ по модулю n, пока не начнётся цикл.

Таблица 9 - coset(i, n)

```
def coset(i, n):
```

```
    """
```

Циклотомический класс по модулю n:

```
    { i, 2i, 4i, 8i, ... } mod n
```

Пока элементы не начнут повторяться.

```
    """
```

```
    c, x = [], i % n
```

```
    while x not in c:
```

```
        c.append(x)
```

```
        x = (2 * x) % n
```

```
    return c
```

Функция `minimal_poly(cos, exp, log, n)` (таблица 10) вычисляет минимальный многочлен над $GF(2)$ для корней α^e , где e принадлежит циклотомическому классу cos , перемножая $(x + \alpha^e)$ в поле $GF(2^m)$ и проверяя, что результат имеет коэффициенты в $GF(2)$.

Таблица 10 - `minimal_poly(cos, exp, log, n)`

```
def minimal_poly(cos, exp, log, n):
```

```
"""
```

Минимальный многочлен над $GF(2)$ для корней $\{\alpha^e : e \in cos\}$.

Строим $\prod(x + \alpha^e)$ в $GF(2^m)$.

Для наших маленьких n результат действительно имеет коэффициенты 0/1.

```
"""
```

```
poly = [1] # в  $GF(2^m)$ 
```

```
for e in cos:
```

```
    a = exp[e] # a =  $\alpha^e$ 
```

```
    poly = pmul_gf(poly, [a, 1], exp, log, n) # (x + a)
```

```
# Проверяем, что коэффициенты "упали" в  $GF(2)$  (0 или 1)
```

```
for c in poly:
```

```
    if c not in (0, 1):
```

```
        raise ValueError("Минимальный многочлен не в  $GF(2)$  (неожиданно для n<=15).")
```

```
return trim(poly)
```

4 Построение БЧХ-кода

Функция `build_bch(n, d)` (таблица 11) основная функция построения БЧН-кода: вычисляет m , создаёт таблицы $GF(2^m)$, находит циклотомические классы, строит минимальные многочлены, вычисляет порождающий многочлен $g(x)$ (как НОК минимальных), проверочный многочлен $h(x)$, определяет параметры k, r , строит порождающую матрицу G и проверочную матрицу H .

Таблица 11 – `build_bch(n, d)`

```
def build_bch(n, d):
    """
    Возвращаем:
        exp, log (для поля)
        g(x), h(x)
        G, H (матрицы)
        k (длина сообщения), r=n-k
        info: список (i, coset, M_i)
    """

    m = int(round(math.log2(n + 1)))
    exp, log, _ = gf_tables(m)

    used = set() # чтобы не брать одинаковые классы дважды
    g = [1] # стартовый g(x)=1
    info = []

    # Берём i = 1..d-1, но фактически учитываем только уникальные циклотомические
    # классы
    for i in range(1, d):
        if i in used:
            continue
        cs = coset(i, n)
        used |= set(cs)
        mp = minimal_poly(cs, exp, log, n)
        g = pmul2(g, mp)
        info.append((i, cs, mp))
```

```

# h(x) = (x^n + 1)/g(x)
xn1 = [1] + [0] * (n - 1) + [1]
h, rem = pdiv2(xn1, g)
if rem != [0]:
    raise RuntimeError("Ошибка: (x^n-1)/g(x) делится с остатком.")

r = len(g) - 1
k = n - r

# Порождающая матрица G: строки - сдвиги g(x) (без циклического переноса)
G = [[0] * n for _ in range(k)]
for i in range(k):
    for j, c in enumerate(g):
        if c:
            G[i][i + j] = 1

# Проверочная матрица H: сдвиги обратного (reciprocal) h(x)
hs = list(reversed(h))
H = [[0] * n for _ in range(r)]
for i in range(r):
    for j, c in enumerate(hs):
        if c:
            H[i][i + j] = 1

return exp, log, g, h, G, H, k, r, info

```

5 Систематическое кодирование

Функция `encode(msg, g, n, k)` (таблица 12) выполняет систематическое кодирование сообщения `msg`: сдвигает его влево на `r` битов, вычисляет остаток от деления на `g(x)`, и добавляет этот остаток к сдвинутому сообщению, формируя кодовое слово `c`.

Таблица 12 - `encode(msg, g, n, k)`

```
def encode(msg, g, n, k):
    r = n - k
    shifted = [0] * r + msg[:]      #  $x^r * m(x)$ 
    rem = pdiv2(shifted, g)[1]      # остаток от деления на  $g(x)$ 
    c = shifted[:]
    for i in range(len(rem)):
        c[i] ^= rem[i]            # добавляем остаток в младшие степени
    return c[:n]
```

6 ПГЦ-декодирование

Функция eval_bin_poly(bits, a, exp, log, n) (таблица 13) вычисляет значение полинома, представленного битами bits, в точке a, где a — элемент поля GF(2^m), используя gf_mul.

Таблица 13 - eval_bin_poly(bits, a, exp, log, n)

```
def eval_bin_poly(bits, a, exp, log, n):
    """Подставляем a в двоичный полином: sum bits[j]*a^j."""
    res, p = 0, 1
    for b in bits:
        if b:
            res ^= p
            p = gf_mul(p, a, exp, log, n)
    return res
```

Функция syndromes(rbits, t, exp, log, n) (таблица 14) вычисляет синдромы S1, S2, ..., S_{2t} для принятого вектора rbits, подставляя α^i в полином rbts(x).

Таблица 14 - syndromes(rbits, t, exp, log, n)

```
def syndromes(rbits, t, exp, log, n):
    """S1..S_{2t} = r(\alpha^1), r(\alpha^2), ..."""
    return [eval_bin_poly(rbits, exp[i], exp, log, n) for i in range(1, 2 * t + 1)]
```

Функция solve_gf(A, b, exp, log, n) (таблица 15) решает систему линейных уравнений $A * x = b$ в поле GF(2^m) с помощью метода Гаусса, используя gf_mul и gf_inv.

Таблица 15 - solve_gf(A, b, exp, log, n)

```
def solve_gf(A, b, exp, log, n):
    """
    Решаем A*x=b в GF(2^m) методом Гаусса.
    Здесь A квадратная v×v. Если pivot не найден -> нет решения/вырождение.
    """

    v = len(A)
    M = [A[i][:] + [b[i]] for i in range(v)]

    for col in range(v):
        piv = None
        for row in range(col, v):
```

```

if M[row][col] != 0:
    piv = row
    break

if piv is None:
    return None

M[col], M[piv] = M[piv], M[col]

invp = gf_inv(M[col][col], exp, log, n)
for j in range(col, v + 1):
    M[col][j] = gf_mul(M[col][j], invp, exp, log, n)

for row in range(v):
    if row == col:
        continue
    f = M[row][col]
    if f:
        for j in range(col, v + 1):
            M[row][j] ^= gf_mul(f, M[col][j], exp, log, n)

return [M[i][v] for i in range(v)]

```

Функция `pgz_decode(rbits, n, d, exp, log)` (таблица 16) реализует классический алгоритм PGZ: вычисляет синдромы, перебирает возможное число ошибок v , составляет и решает систему уравнений для коэффициентов многочлена локаторов $\sigma(x)$, находит позиции ошибок как корни $\sigma(x)$, инвертирует биты в найденных позициях для исправления ошибок.

Таблица 16 - `pgz_decode(rbits, n, d, exp, log)`

```

def pgz_decode(rbits, n, d, exp, log):
    t = (d - 1) // 2
    if t == 0:
        return rbits[:, :, :]

```

```

S = syndromes(rbits, t, exp, log, n)

if all(s == 0 for s in S):
    return rbits[:, :], S

# Подбираем реальное число ошибок v (обычно v<=t).
# Пытаемся v=t, t-1, ..., 1 и ищем решаемую систему.
sigma = None
v_found = 0

# Система PGZ:
#  $S_{\{v+k\}} + \sigma_1 S_{\{v+k-1\}} + \dots + \sigma_v S_{\{k\}} = 0$ ,  $k=1..v$ 
#  $\Rightarrow [S_{\{v+k-1\}} \dots S_{\{k\}}] * [\sigma_1 \dots \sigma_v]^T = S_{\{v+k\}}$ 

for v in range(t, 0, -1):
    A, b = [], []
    ok = True

    for krow in range(1, v + 1):
        row = []
        for j in range(1, v + 1):
            idx = v + krow - j # индекс синдрома
            if not (1 <= idx <= 2 * t):
                ok = False
                break
            row.append(S[idx - 1])
        if not ok:
            break
        rhs = v + krow
        if not (1 <= rhs <= 2 * t):
            ok = False
            break
        A.append(row)
        b.append(S[v+krow])
    if ok:
        sigma = np.linalg.solve(A, b)
        v_found = v
        break

```

```

A.append(row)

b.append(S[rhs - 1])

if not ok:

    continue


sol = solve_gf(A, b, exp, log, n)

if sol is None:

    continue


sigma = [1] + sol[:] # σ(x)=1+σ1 x+...+σv x^v

v_found = v

break


if sigma is None:

    # Не удалось: возможно ошибок > t

    return rbits[:, :], S


# Проверяем σ(α^{-j})=0 для j=0..n-1.

# α^{-j} = α^{n-j} (поскольку α^n = 1)

def sigma_eval(z):

    acc, p = 0, 1

    for c in sigma:

        if c:

            acc ^= gf_mul(c, p, exp, log, n)

            p = gf_mul(p, z, exp, log, n)

    return acc


err_pos = []

for j in range(n):

    z = exp[(n - j) % n] # α^{-j}

```

```
if sigma_eval(z) == 0:  
    err_pos.append(j)  
  
# Исправляем найденные ошибки: flip соответствующие биты  
corrected = rbits[:]  
for j in err_pos:  
    corrected[j] ^= 1  
  
return corrected, err_pos, S
```

7 Ввод/вывод

Функция `read_int(prompt, okset)` (таблица 17) читает целое число с клавиатуры, проверяя, входит ли оно в разрешённый набор `okset` (если задан).

Таблица 17 - `read_int(prompt, okset)`

```
def read_int(prompt, okset=None):
    while True:
        try:
            x = int(input(prompt).strip())
            if okset and x not in okset:
                print(f"Можно только: {sorted(okset)}")
                continue
            return x
        except:
            print("Нужно целое число.")
```

Функция `read_bits(prompt, L)` (таблица 18) читает строку из `L` двоичных символов ('0' или '1') с клавиатуры и возвращает их как список целых [0, 1, ...].

Таблица 18 - `read_bits(prompt, L)`

```
def read_bits(prompt, L):
    while True:
        s = input(prompt).strip().replace(" ", "")
        if len(s) != L or any(ch not in "01" for ch in s):
            print(f"Нужно ровно {L} бит (0/1).")
            continue
        return [int(ch) for ch in s]
```

Функция `print_mat(M, name)` (таблица 19) выводит матрицу `M` на экран с заголовком `name`.

Таблица 19 - `print_mat(M, name)`

```
def print_mat(M, name):
    print(f"\n{name} ({len(M)}x{len(M[0])}):")
    for row in M:
        print(" ".join(map(str, row)))
```

8. Основной код программы (main)

В основном коде поступают ключевые переменные от пользователя и вызываются функции.

Таблица 20 – основной код main

```
if __name__ == "__main__":
    print("==> BCH-код (n=3/7/15) + кодирование + PGZ-декодирование ==>")

    # Ввод параметров кода
    n = read_int("Введите n (3, 7 или 15): ", {3, 7, 15})
    d = read_int(f"Введите d (2..{n}): ")
    if d < 2 or d > n or d > 15:
        raise SystemExit("Некорректное d (должно быть 2..n и <=15).")

    # Строим код (g,h,G,H и вспомогательные данные)
    exp, log, g, h, G, H, k, r, info = build_bch(n, d)

    print(f"\nПараметры кода: n={n}, k={k}, r={r}, d(designed)={d}, t={(d-1)//2}")
    print(f"Порождающий многочлен g(x): {pstr(g)}")
    print(f"Проверочный многочлен h(x): {pstr(h)}")

    # Показываем, какие циклотомические классы реально вошли в g(x)
    print("\nЦиклотомические классы (используются в построении g):")
    for i0, cs, mp in info:
        print(f" i={i0}: класс {cs} => M_{i0}(x) = {pstr(mp)}")

    # Матрицы
    print_mat(G, "Порождающая матрица G")
    print_mat(H, "Проверочная матрица H")

    # ---- Кодирование ----
    print("\n--- КОДИРОВАНИЕ ---")
    print("Систематическое кодирование: c(x)=x^r m(x)+остаток(x^r m(x) mod g(x))")
    msg = read_bits(f"Введите сообщение m (k={k} бит): ", k)
```

```

code = encode(msg, g, n, k)

print("\nКодовое слово выводим в двух видах:")
print(" 1) c0..c_{n-1} (степени от 0 к n-1):", "".join(map(str, code)))
print(" 2) c_{n-1}..c0 (привычная запись строкой):", "".join(map(str, reversed(code))))
print(f"В виде c0..c_{n-1}: первые r={r} бит — проверочные, затем k={k} бит — сообщение.")

# ---- Декодирование ----
print("\n--- ДЕКОДИРОВАНИЕ (PGZ) ---")
print("Введите принятое слово длины n.")
print("Если хотите проверить исправление — внесите до t ошибок (t=floor((d-1)/2)).")
recv = read_bits(f"Принятое слово (n={n} бит): ", n)

corr, pos, S = pgz_decode(recv, n, d, exp, log)

print("\nСиндромы (S1..S_{2t}):")
if S:
    print(" " + " ".join(f"S{i+1}={S[i]}" for i in range(len(S))))
else:
    print(" t=0, синдромы не считаются.")

if pos:
    print("\nPGZ нашёл позиции ошибок j (в порядке коэффициентов c0..):", pos)
    print("Исправление = инверсия битов в этих позициях.")
else:
    print("\nОшибка не найдено ИЛИ ошибок больше, чем t (PGZ не справился).")

print("\nИсправленное слово:")
print(" c0..c_{n-1}:", "".join(map(str, corr)))
print(" c_{n-1}..c0:", "".join(map(str, reversed(corr))))
# Так как кодирование систематическое, сообщение лежит в позициях r..n-1
msg_hat = corr[r:r+k]
print("\nДекодированное сообщение (k бит):", "".join(map(str, msg_hat)))

```

ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы был разработан программный модуль для построения, кодирования и декодирования двоичных кодов БЧХ при длине кода $n \leq 15$. Реализованы ключевые компоненты теории кодирования: построены порождающая и проверочная матрицы, вычислены порождающий и проверочный многочлены, а также обеспечено систематическое кодирование информационных сообщений. Декодирование осуществлялось с использованием классического алгоритма Питерсона–Горенстейна–Цирлера (ПГЦ), способного определять и исправлять до $t = \lfloor (d-1)/2 \rfloor$ ошибок, где d - заданное конструктивное минимальное кодовое расстояние. Программа корректно работает с полями Галуа $GF(2^m)$ используя таблицы экспонент и логарифмов для эффективного выполнения арифметических операций.

Практическая реализация подтвердила теоретические свойства БЧХ-кодов: при внесении в кодовое слово числа ошибок, не превышающего корректирующую способность кода, алгоритм ПГЦ успешно обнаруживает позиции ошибок и восстанавливает исходное сообщение. Разработанное решение является гибким — оно поддерживает различные значения n (3, 7, 15) и d , что позволяет исследовать поведение кода при разных параметрах и оценивать компромисс между избыточностью и помехоустойчивостью. Таким образом, лабораторная работа не только закрепила теоретические знания по алгебраическим кодам, но и продемонстрировала их практическую применимость в системах передачи данных с защитой от ошибок.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Русскоязычная документация по python // pylessons.readthedocs.io. – URL:
<https://pylessons.readthedocs.io/ru/latest/basics/operators.html>
(дата обращения: 16.12.2025).
2. Лекции Давыдова Вадима Валерьевича
(дата обращения: 16.12.2025).
3. ДЕКОДЕР ПИТЕРСОНА-ГОРЕНСТЕЙНА-ЦИРЛЕРА //scask.ru. – URL:
https://scask.ru/h_book_tpc.php?id=44
(дата обращения: 16.12.2025).

ПРИЛОЖЕНИЕ А

Модель системы помехоустойчивой связи

```
import math

PRIM = {1: 0b11, 2: 0b111, 3: 0b1011, 4: 0b10011}

def gf_tables(m):
    n = 2**m - 1
    prim = PRIM[m]
    exp = [0] * (2 * n)
    log = [0] * 2**m
    exp[0] = 1
    log[0] = -1
    log[1] = 0
    for i in range(1, n):
        x = exp[i - 1] << 1
        if x & (1 << m):
            x ^= prim
        exp[i] = x & ((1 << m) - 1)
        log[exp[i]] = i
    for i in range(n, 2 * n):
        exp[i] = exp[i - n]
    return exp, log, n

def gf_mul(a, b, exp, log, n):
    if a == 0 or b == 0: return 0
    return exp[(log[a] + log[b]) % n]

def gf_inv(a, exp, log, n):
    if a == 0: raise ZeroDivisionError("Обратный элемент для 0 не существует")
    return exp[-log[a]]

def trim(p):
    while len(p) > 1 and p[-1] == 0: p.pop()
    return p

def pmul2(p, q):
    r = [0] * (len(p) + len(q) - 1)
    for i, a in enumerate(p):
        for j, b in enumerate(q):
            r[i + j] += a * b
    return r
```

```

if a:
    for j, b in enumerate(q):
        if b: r[i + j] ^= 1
return trim(r)

def pdiv2(dividend, divisor):
    dd = trim(dividend[:])
    dv = trim(divisor[:])
    q = [0] * max(0, len(dd) - len(dv) + 1)
    while len(dd) >= len(dv) and dd != [0]:
        sh = len(dd) - len(dv)
        q[sh] = 1
        for i, c in enumerate(dv):
            if c: dd[i + sh] ^= 1
        dd = trim(dd)
    return trim(q), trim(dd)

def pstr(p):
    t = [f"'1' if i==0 else ('x' if i==1 else f'x^{i}')}" for i, c in enumerate(p) if c]
    return "0" if not t else " + ".join(reversed(t))

def pmul_gf(p, q, exp, log, n):
    r = [0] * (len(p) + len(q) - 1)
    for i, a in enumerate(p):
        if a:
            for j, b in enumerate(q):
                if b: r[i + j] ^= gf_mul(a, b, exp, log, n)
    return r

def coset(i, n):
    c, x = [], i % n
    while x not in c:
        c.append(x)
        x = (2 * x) % n
    return c

def minimal_poly(cos, exp, log, n):
    poly = [1]

```

```

for e in cos:
    a = exp[e]
    poly = pmul_gf(poly, [a, 1], exp, log, n)
for c in poly:
    if c not in (0, 1): raise ValueError("Минимальный многочлен не в GF(2).")
    return trim(poly)

def build_bch(n, d):
    m = int(round(math.log2(n + 1)))
    exp, log, _ = gf_tables(m)
    used = set()
    g = [1]
    info = []
    for i in range(1, d):
        if i in used: continue
        cs = coset(i, n)
        used.update(cs)
        mp = minimal_poly(cs, exp, log, n)
        g = pmul2(g, mp)
        info.append((i, cs, mp))
    xn1 = [1] + [0] * (n - 1) + [1]
    h, rem = pdiv2(xn1, g)
    if rem != [0]: raise RuntimeError("Ошибка:  $(x^{n-1})/g(x)$  делится с остатком.")
    r = len(g) - 1
    k = n - r
    G = [[0] * n for _ in range(k)]
    for i in range(k):
        for j, c in enumerate(g):
            if c: G[i][i + j] = 1
    hs = list(reversed(h))
    H = [[0] * n for _ in range(r)]
    for i in range(r):
        for j, c in enumerate(hs):
            if c: H[i][i + j] = 1

```

```

return exp, log, g, h, G, H, k, r, info

def encode(msg, g, n, k):
    r = n - k
    shifted = [0] * r + msg[:]
    rem = pdiv2(shifted, g)[1]
    c = shifted[:]
    for i in range(len(rem)): c[i] ^= rem[i]
    return c[:n]

def eval_bin_poly(bits, a, exp, log, n):
    res, p = 0, 1
    for b in bits:
        if b: res ^= p
        p = gf_mul(p, a, exp, log, n)
    return res

def syndromes(rbits, t, exp, log, n):
    return [eval_bin_poly(rbits, exp[i], exp, log, n) for i in range(1, 2 * t + 1)]

def solve_gf(A, b, exp, log, n):
    v = len(A)
    M = [A[i][:] + [b[i]] for i in range(v)]
    for col in range(v):
        piv = next((row for row in range(col, v) if M[row][col] != 0), None)
        if piv is None: return None
        M[col], M[piv] = M[piv], M[col]
        invp = gf_inv(M[col][col], exp, log, n)
        for j in range(col, v + 1): M[col][j] = gf_mul(M[col][j], invp, exp, log, n)
        for row in range(v):
            if row == col: continue
            f = M[row][col]
            if f:
                for j in range(col, v + 1): M[row][j] ^= gf_mul(f, M[col][j], exp, log, n)
    return [M[i][v] for i in range(v)]

```

```

def pgz_decode(rbits, n, d, exp, log):
    t = (d - 1) // 2
    if t == 0: return rbits[:, [], []]
    S = syndromes(rbits, t, exp, log, n)
    if all(s == 0 for s in S): return rbits[:, [], S]
    sigma = None
    for v in range(t, 0, -1):
        A, b = [], []
        ok = True
        for krow in range(1, v + 1):
            row = [S[v + krow - j - 1] for j in range(1, v + 1)]
            if any(not (1 <= (v + krow - j) <= 2 * t) for j in range(1, v + 1)) or \
                not (1 <= (v + krow) <= 2 * t): ok = False; break
            A.append(row)
            b.append(S[v + krow - 1])
        if not ok: continue
        sol = solve_gf(A, b, exp, log, n)
        if sol is not None:
            sigma = [1] + sol[:, :]
            break
    if sigma is None: return rbits[:, [], S]
    def sigma_eval(z):
        acc, p = 0, 1
        for c in sigma:
            if c: acc ^= gf_mul(c, p, exp, log, n)
            p = gf_mul(p, z, exp, log, n)
        return acc
    err_pos = [j for j in range(n) if sigma_eval(exp[(n - j) % n]) == 0]
    corrected = rbits[:, :]
    for j in err_pos: corrected[j] ^= 1
    return corrected, err_pos, S
def read_int(prompt, okset=None):
    while True:

```

```

try:
    x = int(input(prompt).strip())
    if okset and x not in okset: print(f"Можно только: {sorted(okset)}"); continue
    return x
except: print("Нужно целое число.")

def read_bits(prompt, L):
    while True:
        s = input(prompt).strip().replace(" ", "")
        if len(s) != L or any(ch not in "01" for ch in s): print(f"Нужно ровно {L} бит (0/1)."); continue
        return [int(ch) for ch in s]

def print_mat(M, name):
    print(f"\n{name} ({len(M)}x{len(M[0])}):")
    for row in M: print(" ".join(map(str, row)))

if __name__ == "__main__":
    print("==== BCH-код (n=3/7/15) + кодирование + PGZ-декодирование ====")
    n = read_int("Введите n (3, 7 или 15): ", {3, 7, 15})
    d = read_int(f"Введите d (2..{n}): ")
    if d < 2 or d > n or d > 15: raise SystemExit("Некорректное d.")
    exp, log, g, h, G, H, k, r, info = build_bch(n, d)
    print(f"\nПараметры кода: n={n}, k={k}, r={r}, d(designed)={d}, t={(d-1)//2}")
    print(f"g(x): {pstr(g)}\nh(x): {pstr(h)}")
    print("\nЦиклотомические классы:")
    for i0, cs, mp in info: print(f" i={i0}: {cs} => M_{i0}(x) = {pstr(mp)}")
    print_mat(G, "G")
    print_mat(H, "H")
    print("\n--- КОДИРОВАНИЕ ---")
    msg = read_bits(f"Введите сообщение m (k={k} бит): ", k)
    code = encode(msg, g, n, k)
    print("\nКодовое слово:")
    print(" c0..c_{n-1}: ", "".join(map(str, code)))
    print(" c_{n-1}..c0: ", "".join(map(str, reversed(code))))
    print(f"Проверочные: {code[:r]}, Сообщение: {code[r:]}.")

```

```
print("\n--- ДЕКОДИРОВАНИЕ (PGZ) ---")
recv = read_bits(f"Принятое слово (n={n} бит): ", n)
corr, pos, S = pgz_decode(recv, n, d, exp, log)
print("\nСиндромы (S1..S_{2t}):")
if S: print(" " + " ".join(f"S{i+1}={S[i]}" for i in range(len(S))))
else: print(" t=0.")
if pos: print("\nПозиции ошибок:", pos)
else: print("\nОшибка не найдено ИЛИ PGZ не справился.")
print("\nИсправленное слово:")
print(" c0..c_{n-1}:", "".join(map(str, corr)))
print(" c_{n-1}..c0:", "".join(map(str, reversed(corr))))
msg_hat = corr[r:r+k]
print("\nДекодированное сообщение (k бит):", "".join(map(str, msg_hat)))
```