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		В.В. Давыдов
должность, уч. степень, звание	подпись, дата	инициалы, фамилия

ЛАБОРАТОРНАЯ РАБОТА №1

«МОДЕЛИРОВАНИЕ СИСТЕМЫ ПОМЕХОУСТОЙЧИВОЙ СВЯЗИ. ЧАСТЬ ПЕРВАЯ»

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 3. По заданным с клавиатуры параметрам (n, k) программный модуль должен построить линейный блоковый двоичный код, исправляющий одну ошибку. Если для заданных параметров такого кода не существует, должна быть возвращена ошибка. Для найденного кода на экран выводится его проверочная матрица, а также количество ошибок, которые код может исправить. При построении кода нужно проверить границы Хэмминга, Варшамова-Гилберта и Синглтона. Также модуль должен поддерживать кодирование информации и её декодирование с помощью синдромов.

Теоретические сведения

Граница Хэмминга (Верхняя граница):

$$N \leq \frac{q^n}{\sum_{i=0}^{t} \binom{n}{i}(q-1)^i}$$

Граница Хэмминга определяет пределы возможных значений параметров произвольного блокового кода. Также известна как граница сферической упаковки. Коды, достигающие границы Хэмминга, называют совершенными. Обязательная граница, определяет возможность существования кода (не существует кода, в котором число слов больше, чем верхняя граница).

Граница Варшамова – Гилберта (Нижняя граница):

$$N \geq \frac{q^n}{\sum_{i=0}^{d-1} \binom{n}{i}(q-1)^i}$$

Граница Варшамова – Гилберта определяет, что точно существует код, число слов в котором больше, чем нижняя граница. Не обязательная граница, не определяет возможность существования кода.

Граница Синглтона:

$$d \leq n - k + 1$$

Граница Синглтона – это верхняя оценка минимального кодового расстояния d для любого линейного блокового кода длины n с числом информационных символов k. Обязательная граница, определяет возможность существования кода.

Порождающая матрица G:

Порождающая матрица G линейного (n,k)-кода — это бинарная матрица размера k×n, строки которой образуют базис кодового подпространства. Любое кодовое слово с длины n может быть получено как линейная комбинация строк. Систематический вид:

$$G = [I_{(k)}|A]$$

I – единичная матрица размера k×k,

A – матрица размера k×r, где r = n−k — количество проверочных битов.

В лабораторной работе используется для кодирования слова $c=u*G$, построения матрицы H в систематическом виде.

Проверочная матрица H:

Проверочная матрица H — это матрица размера (n-k)×n, удовлетворяющая условию:

$$H=[-A^T|I_{(n-k)}]$$

A^T — транспонированная матрица A,

$I_{(n-k)}$ — единичная матрица размера r×r.

В лабораторной работе используется для декодирования принятого слова: поиска синдрома, вектора ошибки *e*.

Синдром:

r – принятое слово,

H – проверочная матрица (n−k)×n линейного (n,k)-кода.

Тогда синдром s вычисляется по формуле:

$$s = r \cdot H^T \pmod 2$$

Синдром служит вспомогательным вектором для декодирования принятого слова, сравнивая синдром со столбцами проверочной матрицей H – находим позицию, в которой была допущена ошибка (вектор *e*).

Алгоритм декодирования принятого слова:

Декодирование принятого слова осуществляется следующим алгоритмом:

- Поиск синдрома S:

Принятое слово (received) * H^T

- Если синдром равен нулю, то ошибок нет, слово декодируется путем извлечение информационных *k*-бит.

- Иначе определяется на какой позиции произошла ошибка (поиск *e*) путем проверки на совпадение столбцов в матрице H и получившегося синдрома S (только для двоичных кодов, исправляющих 1 ошибку).

- При обнаружении позиции ошибки – декодируем слово путем инвертации бита на месте ошибки (только для двоичных кодов).

Алгоритм кодирования принятого слова:

Кодирование информационного слова u осуществляется путем умножения вектора – строки u (1xk) на матрицу G(kxn).

$$c = u * G$$

Ход работы

1 Проверка границ Хэмминга, Варшамова-Гилберта и Синглтона

В первую очередь нужно проверить возможность существования заданного n, k кода при помощи границ Хэмминга, Синглтона и Варшамова-Гилберта.

Таблица 1 – Функция проверки границ

```
def check_bounds(n, k, t=1, d=3):
    r = n - k  # Количество проверочных битов
    if r <= 0:
        return False, False, False

    # Граница Хэмминга:
    hamming = 2 ** k <= 2 ** n / (1+n)

    # Граница Синглтона:
    singleton = (n - k) >= (d - 1)

    # Граница Варшамова-Гилберта:
    vg_sum = sum(math.comb(n - 1, i) for i in range(d - 1))
    vg = vg_sum < (2 ** r)

    return hamming, singleton, vg
```

2 Генерация порождающей матрицы G

Если условия существования кода выполнены, то генерируем порождающую матрицу G (kxn) в систематическом виде G = [I|A].

Таблица 2 – Функция генерации порождающей матрицы G

```
def generate_G(n, k):
    r = n - k
    if r < 2:  # Для исправления 1 ошибки требуется d=3, что требует r>=2
        return None
    # Генерация всех ненулевых двоичных векторов длины r
    all_vectors = []
    for i in range(1, 2 ** r):
        vec = [int(bit) for bit in bin(i)[2:].zfill(r)]
        all_vectors.append(vec)

    # Исключение базисных векторов (с ровно одной единицей)
    non_basis_vectors = []
    ones_per_row = [row.count(1) for row in all_vectors]
    for i in range(len(ones_per_row)):
        if ones_per_row[i] != 1:
            non_basis_vectors.append(all_vectors[i])

    # Формируем матрицу A (k x r), где каждая строка - выбранный вектор (до к)
    A = non_basis_vectors[:k]

    # Строим порождающую матрицу G = [I_k | A]
    G = []
    for i in range(k):
        row = [0] * n
        row[i] = 1  # Единичная часть
        for j in range(r):
            row[k + j] = A[i][j]  # Часть A
        G.append(row)

    return G
```

3 Построение проверочной матрицы H из порождающей матрицы G

Из порождающей матрицы G(kxn) строим проверочную матрицу H ((n-k)xn) путем транспонирования матрицы A и добавлением справа единичной матрицы I (n-k), H = [A^T|I].

Таблица 3 – Функция построения проверочной матрицы H

```
def get_H_from_G(G, n, k):
    r = n - k
    # Извлекаем матрицу A из G (правая часть после единичной матрицы)
    A = [row[k:] for row in G]  # A имеет размер k × r

    # Транспонируем A; A_T будет размером r × k
    A_T = []
    for j in range(r):          # по каждому столбцу A
        new_row = []
        for i in range(k):      # по каждой строке A
            new_row.append(A[i][j])
        A_T.append(new_row)

    # Формируем H = [A^T | I_r] с помощью циклов
    H = []
    for i in range(r):
        # Создаём новую строку длины n (поскольку H имеет r строк и n столбцов)
        row = [0] * n

        # Заполняем левую часть (A^T): первые k столбцов
        for j in range(k):
            row[j] = A_T[i][j]

        # Заполняем правую часть (I_r): столбцы с k до n-1
        for j in range(r):
            if j == i:
                row[k + j] = 1
            # иначе остаётся 0 (уже установлено при инициализации)
        H.append(row)
    return H
```

4 Кодирование информационного слова с помощью порождающей матрицы G

Кодирование информационного слова u осуществляется путем умножения вектора – строки u (1xk) на матрицу G.

Таблица 4 – Функция кодирования информационного слова

```
def encode(u, G, n, k):
    codeword = [0] * n
    for i in range(k):
        if u[i] == 1:
            for j in range(n):
                codeword[j] = (codeword[j] + G[i][j]) % 2
    return codeword
```

5 Декодирование принятого слова при помощи синдрома и проверочной матрицы H

Декодирование принятого слова осуществляется следующим алгоритмом:

1) Поиск синдрома S:
   Принятое слово (received) * H^T
2) Если синдром равен нулю, то ошибок нет, слово декодируется путем извлечение информационных к-бит
3) Иначе определяется на какой позиции произошла ошибка (поиск e) путем проверки на совпадение столбцов в матрице H и получившегося синдрома S
4) При обнаружении позиции ошибки – декодируем слово путем инвертации бита на месте ошибки

Таблица 5 – Функция декодирования принятого слова

```
def decode(received, H, n, k):
    r = n - k
    # Вычисляем синдром: s = received * H^T
    syndrome = [0] * r
    for i in range(r):
        total = 0
        for j in range(n):
            total = (total + received[j] * H[i][j]) % 2
        syndrome[i] = total


    # Если синдром ненулевой, ищем позицию ошибки
```

```
    error_pos = None
    if any(syndrome):  #Если синдром не ноль
        # Проверяем каждый столбец H на совпадение с синдромом
        for j in range(n):
            column = [H[i][j] for i in range(r)]  #Транспонировали H
            if column == syndrome:
                error_pos = j
                break

    # Исправляем ошибку, если она найдена
    corrected = received.copy()
    if error_pos is not None:
        corrected[error_pos] = (corrected[error_pos] + 1) % 2

    # Извлекаем информационные биты (первые k бит в систематическом виде)
    info_bits = corrected[:k]
    return info_bits, syndrome, error_pos
```

6 Вывод матрицы в удобочитаемом формате

Выводим получившиеся матрицы.

Таблица 6 – Функция вывода матриц H и G

```
def print_matrix(matrix, name):
    print(f"\n{name} матрица:")
    for row in matrix:
        print(' '.join(str(x) for x in row))
```

7 Основной код программы

Таблица 7 – Основной код программы

```
def main():
    try:
        n = int(input("Введите длину кодового слова (n): "))
        k = int(input("Введите длину информационного слова (k): "))
    except ValueError:
        print("Ошибка: n и k должны быть целыми числами.")
        return
```

```
    if n <= k or k <= 0 or n <= 0:
        print("Ошибка: Некорректные параметры (n > k > 0).")
        return

    # Проверка границ для кода, исправляющего 1 ошибку (d=3)
    hamming, singleton, gv = check_bounds(n, k, t=1, d=3)

    print("\nРезультаты проверки границ:")
    print(f"Граница Хэмминга (2^k <= 2^n / (1+n)): {'✓' if hamming else '✗'}")
    print(f"Граница Синглтона (d ≤ n-k+1): {'✓' if singleton else '✗'}")
    print(f"Граница Варшамова-Гилберта (n < 2^(n-k)): {'✓' if gv else '✗'}")

    if not (hamming and singleton and gv):
        print("Ошибка: Для заданных параметров невозможно построить код, исправляющий одну ошибку.")
        return

    # Генерация порождающей матрицы G
    G = generate_G(n, k)
    if G is None:
        print("Ошибка: Не удалось сгенерировать порождающую матрицу. Проверьте параметры.")
        return

    # Получение проверочной матрицы H из G
    H = get_H_from_G(G, n, k)

    # Вывод матриц
    print_matrix(G, "Порождающая (G = [I|A])")
    print_matrix(H, "Проверочная (H = [A^T|I])")

    print(f"\nКод может исправить 1 ошибку.")

    # Демонстрация кодирования и декодирования
```

```
    try:
        info_input = input(f"\nВведите информационное слово длины {k} (биты 0/1 без пробелов): ")
        if len(info_input) != k or any(c not in '01' for c in info_input):
            raise ValueError("Неверный формат информационного слова")
        u = [int(bit) for bit in info_input]

        # Кодирование
        codeword = encode(u, G, n, k)
        print("Закодированное слово:", ".join(str(bit) for bit in codeword))

        # Имитация передачи с возможной ошибкой
        received_input = input(f'Введите принятое слово длины {n} (биты 0/1 без пробелов): ")
        if len(received_input) != n or any(c not in '01' for c in received_input):
            raise ValueError("Неверный формат принятого слова")
        received = [int(bit) for bit in received_input]

        # Декодирование
        decoded_info, syndrome, error_pos = decode(received, H, n, k)

        print(f"\nСиндром: {syndrome} ")
        if error_pos is not None:
            print(f"Обнаружена и исправлена ошибка в позиции {error_pos + 1}")
        else:
            if any(syndrome):
                print("Обнаружена ошибка, но её позиция не определена (более одной ошибки)")
            else:
                print("Ошибок не обнаружено")

        print("Декодированное информационное слово:", ".join(str(bit) for bit in decoded_info))

    except Exception as e:
        print(f"Ошибка при кодировании/декодировании: {e}")
```

ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы был разработан и реализован программный модуль на языке Python, предназначенный для построения, кодирования и декодирования линейного блокового двоичного кода, способного исправлять одиночные ошибки. Программа охватывает полный цикл работы с корректирующими кодами: от теоретической проверки возможности существования кода с заданными параметрами до практического применения кодирования и синдромного декодирования.

Выводы:

1) Теоретические границы работают на практике: проверка границ Хэмминга, Синглтона и Варшамова–Гилберта позволяет заранее определить возможность построения кода с требуемыми свойствами, что исключает попытки генерации заведомо неработающих параметров.

2) Минимальное расстояние d=3 обеспечивает исправление одной ошибки: Реализованный алгоритм надёжно обнаруживает и корректирует одиночные ошибки, что подтверждается тестированием на различных примерах, включая классический код Хэмминга [7,4,3].

3) Систематический формат упрощает обработку данных: размещение информационных битов в начале кодового слова делает процесс кодирования интуитивно понятным, а извлечение информации при декодировании — тривиальным.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Русскоязычная документация по python // pylessons.readthedocs.io. – URL: https://pylessons.readthedocs.io/ru/latest/basics/operators.html

(дата обращения: 23.11.2025).

2. Лекции Давыдова Вадима Валерьевича

(дата обращения: 23.11.2025).

ПРИЛОЖЕНИЕ А

Модель системы помехоустойчивой связи

```
import math
def check_bounds(n, k, t=1, d=3):
    """Проверяем границы Хэмминга, Синглтона и Варшамова-Гилберта для кода с параметрами n, k."""
    r = n - k  # Количество проверочных битов
    if r <= 0:
        return False, False, False
    # Граница Хэмминга:
    hamming = 2 ** k <= 2 ** n / (1+n)
    # Граница Синглтона:
    singleton = (n - k) >= (d - 1)
    # Граница Варшамова-Гилберта:
    vg_sum = sum(math.comb(n - 1, i) for i in range(d - 1))
    vg = vg_sum < (2 ** r)
    return hamming, singleton, vg
def generate_G(n, k):
    """Генерируем порождающую матрицу G в систематическом виде [I_k | A]."""
    r = n - k
    if r < 2:  # Для исправления 1 ошибки требуется d=3, что требует r>=2
        return None
    # Генерация всех ненулевых двоичных векторов длины r
    all_vectors = []
    for i in range(1, 2 ** r):
        vec = [int(bit) for bit in bin(i)[2:].zfill(r)]
        all_vectors.append(vec)
    # Исключение базисных векторов (с ровно одной единицей)
    non_basis_vectors = []
    ones_per_row = [row.count(1) for row in all_vectors]
    for i in range(len(ones_per_row)):
        if ones_per_row[i] != 1:
            non_basis_vectors.append(all_vectors[i])
    # Формируем матрицу A (k x r), где каждая строка - выбранный вектор (до к)
```

```
    A = non_basis_vectors[:k]
    # Строим порождающую матрицу G = [I_k | A]
    G = []
    for i in range(k):
        row = [0] * n
        row[i] = 1  # Единичная часть
        for j in range(r):
            row[k + j] = A[i][j]  # Часть A
        G.append(row)
    return G
def get_H_from_G(G, n, k):
    """Строим проверочную матрицу H из порождающей матрицы G."""
    r = n - k
    # Извлекаем матрицу A из G (правая часть после единичной матрицы)
    A = [row[k:] for row in G]  # A имеет размер k × r
    # Транспонируем A; A_T будет размером r × k
    A_T = []
    for j in range(r):          # по каждому столбцу A
        new_row = []
        for i in range(k):      # по каждой строке A
            new_row.append(A[i][j])
        A_T.append(new_row)
    # Формируем H = [A^T | I_r] с помощью циклов
    H = []
    for i in range(r):
        # Создаём новую строку длины n (поскольку H имеет r строк и n столбцов)
        row = [0] * n
        # Заполняем левую часть (A^T): первые k столбцов
        for j in range(k):
            row[j] = A_T[i][j]
        # Заполняем правую часть (I_r): столбцы с k до n-1
        for j in range(r):
            if j == i:
                row[k + j] = 1
            # иначе остаётся 0 (уже установлено при инициализации)
```

```
        H.append(row)
    return H
def encode(u, G, n, k):
    """Кодируем информационное слово u с помощью порождающей матрицы G."""
    codeword = [0] * n
    for i in range(k):
        if u[i] == 1:
            for j in range(n):
                codeword[j] = (codeword[j] + G[i][j]) % 2
    return codeword
def decode(received, H, n, k):
    """Декодируем принятое слово с использованием синдромов."""
    r = n - k
    # Вычисляем синдром: s = received * H^T
    syndrome = [0] * r
    for i in range(r):
        total = 0
        for j in range(n):
            total = (total + received[j] * H[i][j]) % 2
        syndrome[i] = total
    # Если синдром ненулевой, ищем позицию ошибки
    error_pos = None
    if any(syndrome):  #Если синдром не ноль
        # Проверяем каждый столбец H на совпадение с синдромом
        for j in range(n):
            column = [H[i][j] for i in range(r)]  #Транспонировали H
            if column == syndrome:
                error_pos = j
                break
    # Исправляем ошибку, если она найдена
    corrected = received.copy()
    if error_pos is not None:
        corrected[error_pos] = (corrected[error_pos] + 1) % 2
    # Извлекаем информационные биты (первые k бит в систематическом виде)
    info_bits = corrected[:k]
```

```
        return info_bits, syndrome, error_pos
    def print_matrix(matrix, name):
        """Выводим матрицу в удобочитаемом формате."""
        print(f"\n{name} матрица:")
        for row in matrix:
            print(' '.join(str(x) for x in row))
    def main():
        try:
            n = int(input("Введите длину кодового слова (n): "))
            k = int(input("Введите длину информационного слова (k): "))
        except ValueError:
            print("Ошибка: n и k должны быть целыми числами.")
            return
        if n <= k or k <= 0 or n <= 0:
            print("Ошибка: Некорректные параметры (n > k > 0).")
            return
        # Проверка границ для кода, исправляющего 1 ошибку (d=3)
        hamming, singleton, gv = check_bounds(n, k, t=1, d=3)
        print("\nРезультаты проверки границ:")
        print(f"Граница Хэмминга (2^k <= 2^n / (1+n)): {'✓' if hamming else '✗'}")
        print(f"Граница Синглтона (d ≤ n-k+1): {'✓' if singleton else '✗'}")
        print(f"Граница Варшамова-Гилберта (n < 2^(n-k)): {'✓' if gv else '✗'}")
        if not (hamming and singleton and gv):
            print("Ошибка: Для заданных параметров невозможно построить код, исправляющий одну ошибку.")
            return
        # Генерация порождающей матрицы G
        G = generate_G(n, k)
        if G is None:
            print("Ошибка: Не удалось сгенерировать порождающую матрицу. Проверьте параметры.")
            return
        # Получение проверочной матрицы H из G
        H = get_H_from_G(G, n, k)
```

```
    # Вывод матриц
    print_matrix(G, "Порождающая (G = [I|A])")
    print_matrix(H, "Проверочная (H = [A^T|I])")
    print(f"\nКод может исправить 1 ошибку.")
    # Демонстрация кодирования и декодирования
    try:
        info_input = input(f"\nВведите информационное слово длины {k} (биты 0/1 без пробелов): ")
        if len(info_input) != k or any(c not in '01' for c in info_input):
            raise ValueError("Неверный формат информационного слова")
        u = [int(bit) for bit in info_input]
        # Кодирование
        codeword = encode(u, G, n, k)
        print("Закодированное слово:", ".join(str(bit) for bit in codeword))
        # Имитация передачи с возможной ошибкой
        received_input = input(f"Введите принятое слово длины {n} (биты 0/1 без пробелов): ")
        if len(received_input) != n or any(c not in '01' for c in received_input):
            raise ValueError("Неверный формат принятого слова")
        received = [int(bit) for bit in received_input]
        # Декодирование
        decoded_info, syndrome, error_pos = decode(received, H, n, k)
        print(f"\nСиндром: {syndrome} ")
        if error_pos is not None:
            print(f"Обнаружена и исправлена ошибка в позиции {error_pos + 1}")
        else:
            if any(syndrome):
                print("Обнаружена ошибка, но её позиция не определена (более одной ошибки)")
            else:
                print("Ошибок не обнаружено")
        print("Декодированное информационное слово:", ".join(str(bit) for bit in decoded_info))
    except Exception as e:
        print(f"Ошибка при кодировании/декодировании: {e}")
```

```
if __name__ == "__main__":
    main()
```