

Методическое пособие по MIMO

June 20, 2025

1 Описание всей системы

Цель: Передать информацию через канал связи с использованием различных методов модуляции и демодуляции.

Блоки системы:

1. Генератор битов
2. Модулятор (BPSK)
3. Генератор канала (Rayleigh fading)
4. Модель канала (MIMO, MISO)
5. Эквалайзер (Zero-Forcing, Alamouti)
6. Демодулятор
7. Расчет BER (Bit Error Rate)
8. Теоретические кривые BER

Блок-схема системы: Псевдокод системы:

1. Инициализация параметров:
 - num_bits: количество битов для передачи
 - num_trials: количество испытаний для метода Монте-Карло
 - snr_db_list: список значений SNR в дБ
2. Для каждого значения SNR в snr_db_list:
 - 2.1. Инициализация переменных для подсчета BER:
 - ber_mimo_sum = 0
 - ber_alamouti_miso_sum = 0
 - 2.2. Для каждого испытания от 1 до num_trials:

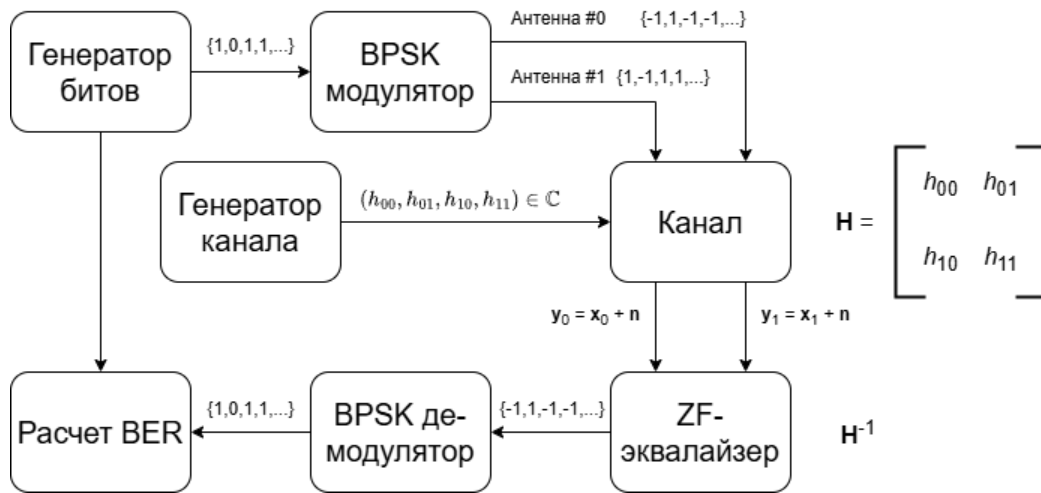


Figure 1: Блок-схема системы

2.2.1. Генерация битов:

- `mimo_bits = generate_bits(2 * num_bits)`
- `alamouti_bits_miso = generate_bits(num_bits)`

2.2.2. Модуляция BPSK:

- `mimo_symbols = bpsk_modulate(mimo_bits)`
- `alamouti_symbols_miso = bpsk_modulate(alamouti_bits_miso)`

2.2.3. Генерация канала:

- `h_mimo = generate_channel_matrix()`
- `h_miso = generate_miso_channel()`

2.2.4. Модель канала:

- `mimoY = mimo_channel(mimo_symbols, h_mimo, SNR)`
- `alamoutiYMISO = miso_channel(alamouti_symbols_miso, h_miso, SNR)`

2.2.5. Демодуляция:

- `mimo_xhat = zf_demodulate(mimoY, h_mimo)`
- `decoded_miso = alamouti_decode_miso(alamoutiYMISO, h_miso)`

2.2.6. Демодуляция BPSK:

- `mimo_bits_hat = bpsk_demodulate(mimo_xhat)`
- `alamouti_bits_hat_miso = bpsk_demodulate(decoded_miso)`

2.2.7. Расчет BER:

- `ber_mimo_sum += calculate_ber(mimo_bits, mimo_bits_hat)`
- `ber_alamouti_miso_sum += calculate_ber(alamouti_bits_miso, alamouti_bits_hat_miso)`

2.3. Усреднение BER:

- `ber_mimo = ber_mimo_sum / num_trials`
- `ber_alamouti_miso = ber_alamouti_miso_sum / num_trials`

2.4. Вывод результатов:

- Вывод BER для MIMO и Alamouti MISO
- Сравнение с теоретическими значениями BER

2 Генератор битов

Описание: Генератор битов создает случайную последовательность битов, которая используется для моделирования передачи данных через канал связи.

Зачем нужен: Для создания исходных данных, которые будут передаваться через канал. Это позволяет моделировать реальные условия передачи данных и оценивать качество связи.

Теоретическое описание: Генерация битов обычно осуществляется с использованием генератора случайных чисел. Каждый бит может принимать значение 0 или 1 с равной вероятностью. Это позволяет создать случайную последовательность битов, которая имитирует реальные данные.

Формула:

$$b_i = \begin{cases} 0, & \text{если } r_i < 0.5 \\ 1, & \text{если } r_i \geq 0.5 \end{cases}$$

где b_i — i -й бит, r_i — случайное число от 0 до 1.

Объяснение: Формула генерации битов использует случайное число r_i , которое равномерно распределено в интервале от 0 до 1. Если r_i меньше 0.5, то бит b_i принимает значение 0. В противном случае, бит b_i принимает значение 1. Это обеспечивает равномерное распределение битов 0 и 1.

Вспомогательный материал:

- **C++:** Использование `std::uniform_int_distribution`, `std::random_device`.
- **Python:** Использование `numpy.random.randint`.

Пример кода на C++:

```
1 #include <iostream>
2 #include <random>
3 int generate_random_bit() {
4     std::random_device rd;
5     std::mt19937 gen(rd());
6     std::uniform_int_distribution<> dis(0, 1);
7     return dis(gen);
8 }
```

```
9 | int main() {  
10 |     int random_bit = generate_random_bit();  
11 |     std::cout << "Generated bit: " << random_bit << std::endl  
    |     ;  
12 |     return 0;  
13 | }
```

Пример кода на Python:

```
1 import numpy as np
2 def generate_random_bit():
3     return np.random.randint(0, 2)
4 if __name__ == "__main__":
5     random_bit = generate_random_bit()
6     print(f"Generated bit: {random_bit}")
```

Сигнатура функции:

```
1 std::vector<int> generate_bits(int num_bits);
```

```
1 def generate_bits(num_bits: int) -> np.ndarray:
```

3 Модулятор (BPSK)

Описание: BPSK (Binary Phase Shift Keying) — это метод модуляции, который преобразует биты в символы.

Зачем нужен: Для преобразования битов в форму, пригодную для передачи через канал. **Теоретическое описание:**

BPSK модуляция преобразует биты в символы, используя две фазы: 0 градусов для бита 0 и 180 градусов для бита 1. Это позволяет передавать информацию через канал с минимальными искажениями.

Формула:

$$s_i = \begin{cases} 1, & \text{если } b_i = 0 \\ -1, & \text{если } b_i = 1 \end{cases}$$

где s_i — i -й символ, b_i — i -й бит.

Объяснение: Формула BPSK модуляции определяет, как каждый бит b_i преобразуется в символ s_i . Если бит равен 0, то символ равен 1. Если бит равен 1, то символ равен -1. Это позволяет передавать информацию через канал с минимальными искажениями.

Вспомогательный материал:

- **C++:** Использование `std::vector`, `std::complex`.
- **Python:** Использование `numpy.where`.

Сигнатура функции:

```
1 std::vector<std::complex<double>> bpsk_modulate(const std::
    vector<int> &bits);
```

```
1 def bpsk_modulate(bits: np.ndarray) -> np.ndarray:
```

4 Генератор канала (Rayleigh fading)

Описание: Генератор канала создает случайные комплексные коэффициенты, моделирующие эффект Релея.

Зачем нужен: Для моделирования реальных условий передачи сигнала.

Теоретическое описание: Эффект Релея моделирует распространение сигнала в условиях многолучевого рассеяния, где амплитуда сигнала подчиняется распределению Релея.

Формула:

$$h = \mathcal{N}(0, \frac{1}{\sqrt{2}}) + j \cdot \mathcal{N}(0, \frac{1}{\sqrt{2}})$$

где h — комплексный коэффициент канала, \mathcal{N} — нормальное распределение.

Объяснение: Комплексный коэффициент канала h моделируется как сумма двух независимых нормальных распределений с нулевым средним и стандартным отклонением $\frac{1}{\sqrt{2}}$. Это позволяет моделировать эффект Релея, который характерен для многолучевого рассеяния.

Вспомогательный материал:

- **C++:** Использование `std::normal_distribution`, `std::complex`.
- **Python:** Использование `numpy.random.normal`.

Сигнатура функции:

```
1 std::complex<double> generate_channel_gain();
```

```
1 def generate_channel_gain() -> complex:
```

5 Модель канала (MIMO, MISO)

Описание: Модель канала описывает, как сигнал проходит через канал с учетом шума и затухания.

Зачем нужен: Для моделирования передачи сигнала через канал.

Теоретическое описание: Модель канала учитывает влияние шума и затухания на передаваемый сигнал. В случае MIMO (Multiple Input Multiple Output) и MISO (Multiple Input Single Output) используются матрицы канала для моделирования влияния нескольких антенн.

5.1 Модель MIMO

Описание: В MIMO системах используются несколько передающих и принимающих антенн. Это позволяет увеличить пропускную способность

и надежность передачи данных. **Формула (MIMO):**

$$\mathbf{Y} = \mathbf{H}\mathbf{X} + \mathbf{N}$$

где:

- \mathbf{Y} — вектор принятых сигналов размером $N_r \times 1$,
- \mathbf{H} — матрица канала размером $N_r \times N_t$,
- \mathbf{X} — вектор переданных сигналов размером $N_t \times 1$,
- \mathbf{N} — вектор шума размером $N_r \times 1$.

Объяснение: Каждый элемент вектора принятых сигналов \mathbf{Y} является линейной комбинацией элементов вектора переданных сигналов \mathbf{X} , умноженных на соответствующие коэффициенты матрицы канала \mathbf{H} , с добавлением шума \mathbf{N} .

Формула в виде СЛАУ:

$$\begin{cases} y_1 = h_{11}x_1 + h_{12}x_2 + n_1 \\ y_2 = h_{21}x_1 + h_{22}x_2 + n_2 \\ \vdots \\ y_{N_r} = h_{N_r 1}x_1 + h_{N_r 2}x_2 + n_{N_r} \end{cases}$$

где y_i — i -й элемент вектора принятых сигналов, h_{ij} — элемент матрицы канала, x_j — j -й элемент вектора переданных сигналов, n_i — i -й элемент вектора шума.

5.2 Модель MISO

Описание: В MISO системах используются несколько передающих антенн и одна принимающая антенна. Это позволяет улучшить качество передачи данных за счет использования пространственного разнесения. **Формула (MISO):**

$$y = \sum_{i=1}^{N_t} h_i x_i + n$$

где:

- y — принятый сигнал,
- h_i — коэффициент канала для i -й передающей антенны,

- x_i — переданный символ с i -й антенны,
- n — шум,
- N_t — количество передающих антенн.

Объяснение: Принятый сигнал y является суммой всех переданных символов x_i , умноженных на соответствующие коэффициенты канала h_i , с добавлением шума n . Это позволяет моделировать влияние нескольких передающих антенн на принимаемый сигнал.

Частный случай MISO 2x1: Рассмотрим частный случай MISO 2x1, где используются две передающие антенны и одна принимающая антенна.

Формула (MISO 2x1):

$$y = h_1 x_1 + h_2 x_2 + n$$

где:

- y — принятый сигнал,
- h_1, h_2 — коэффициенты канала для первой и второй передающих антенн,
- x_1, x_2 — переданные символы с первой и второй антенн,
- n — шум.

Объяснение: Принятый сигнал y является линейной комбинацией переданных символов x_1 и x_2 , умноженных на соответствующие коэффициенты канала h_1 и h_2 , с добавлением шума n . Это позволяет моделировать влияние двух передающих антенн на принимаемый сигнал.

Вспомогательный материал:

- **C++:** Использование `std::vector`, `std::normal_distribution`, `std::complex`.
- **Python:** Использование `numpy.random.normal`, `numpy.ndarray`.

Сигнатура функции (MISO):

```
1 void miso_channel(const std::vector<std::complex<double>> &X,
    std::complex<double> &Y, const std::vector<std::complex<
    double>> &h, double SNRdB);
```

```
1 def miso_channel(X: np.ndarray, h: np.ndarray, SNRdB: float)
    -> np.ndarray:
```


6 Эквалайзер (Zero-Forcing, Alamouti)

Описание: Эквалайзер восстанавливает исходные биты из принятых символов.

Зачем нужен: Для восстановления переданной информации.

Теоретическое описание: Эквалайзер использует различные методы для восстановления исходных символов. Zero-Forcing (ZF) эквалайзер использует обратную матрицу канала для восстановления символов. Декодирование Аламоути использует ортогональные свойства кодов Аламоути для восстановления символов.

6.1 Zero-Forcing Демодуляция

Описание: Zero-Forcing (ZF) демодуляция использует обратную матрицу канала для восстановления символов. Этот метод позволяет устранить влияние интерференции между антеннами.

Формула (Zero-Forcing):

$$\hat{\mathbf{X}} = \mathbf{H}^{-1}\mathbf{Y}$$

где:

- $\hat{\mathbf{X}}$ — оценка переданного сигнала,
- \mathbf{H}^{-1} — обратная матрица канала,
- \mathbf{Y} — принятый сигнал.

Объяснение: Оценка переданного сигнала \mathbf{X}_{hat} вычисляется как произведение обратной матрицы канала \mathbf{H}^{-1} и принятого сигнала \mathbf{Y} . Это позволяет устранить влияние интерференции между антеннами.

Связь с СЛАУ: Zero-Forcing эквалайзер фактически решает систему линейных алгебраических уравнений (СЛАУ), описанных в разделе "Модель канала". Рассмотрим систему уравнений:

$$\mathbf{Y} = \mathbf{H}\mathbf{X} + \mathbf{N}$$

При отсутствии шума ($\mathbf{N} = 0$), система уравнений упрощается до:

$$\mathbf{Y} = \mathbf{H}\mathbf{X}$$

Решение этой системы для \mathbf{X} приводит к:

$$\mathbf{X} = \mathbf{H}^{-1}\mathbf{Y}$$

Таким образом, Zero-Forcing эквалайзер решает эту систему, используя обратную матрицу канала.

Методы решения СЛАУ: Существует несколько методов решения систем линейных алгебраических уравнений:

- **Метод Гаусса (или метод Гаусса-Жордана):** Этот метод использует последовательные преобразования строк для приведения матрицы к треугольному виду, что позволяет легко найти решение.
- **Метод LU-разложения:** Этот метод разлагает матрицу \mathbf{H} на произведение нижней треугольной матрицы \mathbf{L} и верхней треугольной матрицы \mathbf{U} . Решение системы сводится к последовательному решению двух треугольных систем.
- **Метод сингулярного разложения (SVD):** Этот метод разлагает матрицу \mathbf{H} на произведение трех матриц: \mathbf{U} , $\mathbf{\Sigma}$ и \mathbf{V}^* . Метод SVD особенно полезен для численно нестабильных систем.

Обоснование использования матричного вида: Использование матричного вида для решения СЛАУ имеет несколько преимуществ:

- **Компактность:** Матричный вид позволяет компактно записать систему уравнений и легко манипулировать ею.
- **Численная стабильность:** Матричные методы, такие как LU-разложение и SVD, обеспечивают численную стабильность при решении систем уравнений.
- **Эффективность:** Современные библиотеки линейной алгебры (например, LAPACK, NumPy) оптимизированы для работы с матрицами и обеспечивают высокую производительность.

Частный случай ММО 2x2: Рассмотрим частный случай ММО 2x2, где матрица канала \mathbf{H} имеет размер 2×2 .

Матрица канала:

$$\mathbf{H} = \begin{pmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{pmatrix}$$

Обратная матрица: Обратная матрица \mathbf{H}^{-1} вычисляется через определитель:

$$\mathbf{H}^{-1} = \frac{1}{\det(\mathbf{H})} \begin{pmatrix} h_{22} & -h_{12} \\ -h_{21} & h_{11} \end{pmatrix}$$

где $\det(\mathbf{H}) = h_{11}h_{22} - h_{12}h_{21}$.

Восстановление символов:

$$\hat{\mathbf{X}} = \mathbf{H}^{-1}\mathbf{Y} = \frac{1}{\det(\mathbf{H})} \begin{pmatrix} h_{22} & -h_{12} \\ -h_{21} & h_{11} \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$$

Объяснение: Каждый элемент вектора оценки переданного сигнала $\hat{\mathbf{X}}$ является линейной комбинацией элементов вектора принятых сигналов \mathbf{Y} , умноженных на соответствующие элементы обратной матрицы канала \mathbf{H}^{-1} .

Вспомогательный материал:

- **C++:** Использование `std::vector`, `std::complex`, матричные операции.
- **Python:** Использование `numpy.linalg.inv`, `numpy.ndarray`.

Сигнатура функции (Zero-Forcing):

```
1 std::vector<std::complex<double>> zf_demodulate(const std::vector<std::complex<double>> &Y, const std::vector<std::complex<double>>> &H);
```

```
1 def zf_demodulate(Y: np.ndarray, H: np.ndarray) -> np.ndarray:
```

6.2 Декодирование Аламоути (MISO 2x1)

Описание: Декодирование Аламоути использует ортогональные свойства кодов Аламоути для восстановления символов. Этот метод позволяет улучшить качество передачи данных за счет использования пространственного разнесения.

Формула (Alamouti):

$$\hat{s}_1 = h_1^* y_1 + h_2 y_2^*$$

$$\hat{s}_2 = h_2^* y_1 - h_1 y_2^*$$

где:

- \hat{s}_1, \hat{s}_2 — оценки переданных символов,
- h_1, h_2 — коэффициенты канала,
- y_1, y_2 — принятые сигналы.

Нормировка:

$$\hat{s}_1 = \frac{\hat{s}_1}{\|h_1\|^2 + \|h_2\|^2}$$
$$\hat{s}_2 = \frac{\hat{s}_2}{\|h_1\|^2 + \|h_2\|^2}$$

Объяснение: Оценки переданных символов \hat{s}_1 и \hat{s}_2 вычисляются как линейные комбинации принятых сигналов y_1 и y_2 , умноженных на сопряженные коэффициенты канала h_1 и h_2 . Нормировка позволяет учесть влияние шума и улучшить точность оценки.

Вспомогательный материал:

- **C++:** Использование `std::vector`, `std::complex`, матричные операции.
- **Python:** Использование `numpy.linalg.inv`, `numpy.ndarray`.

Сигнатура функции (Zero-Forcing):

```
1 std::vector<std::complex<double>> zf_demodulate(const std::vector<std::complex<double>> &Y, const std::vector<std::vector<std::complex<double>>> &H);
```

```
1 def zf_demodulate(Y: np.ndarray, H: np.ndarray) -> np.ndarray:
```

Сигнатура функции (Alamouti):

```
1 std::vector<std::complex<double>> alamouti_decode_miso(const std::vector<std::complex<double>> &Y, const std::vector<std::complex<double>> &h);
```

```
1 def alamouti_decode_miso(Y: np.ndarray, h: np.ndarray) -> np.ndarray:
```

7 Расчет BER (Bit Error Rate)

Описание: BER — это метрика, показывающая, сколько битов было принято с ошибкой.

Зачем нужен: Для оценки качества передачи.

Теоретическое описание: BER (Bit Error Rate) измеряет долю битов, которые были приняты с ошибкой. Это важная метрика для оценки надежности системы передачи данных.

Формула:

$$\text{BER} = \frac{\text{Количество ошибочных битов}}{\text{Общее количество битов}}$$

Объяснение: BER вычисляется как отношение количества ошибочных битов к общему количеству переданных битов. Это позволяет оценить надежность системы передачи данных.

Метод Монте-Карло: Метод Монте-Карло используется для статистического моделирования. Он позволяет оценить BER, многократно повторяя эксперимент и усредняя результаты.

Псевдокод метода Монте-Карло:

1. Инициализация параметров:
 - numTrials: количество испытаний
 - berSum: сумма BER для всех испытаний
2. Для каждого испытания от 1 до numTrials:
 - Генерация битов и передача через канал
 - Демодуляция и расчет BER для текущего испытания
 - berSum += BER текущего испытания
3. Усреднение BER:
 - ber = berSum / numTrials

Вспомогательный материал:

- C++: Использование `std::vector`, `std::min`.
- Python: Использование `numpy.sum`, `numpy.min`.

Сигнатура функции:

```
1 double calculate_ber(const std::vector<int> &original, const
  std::vector<int> &received);
```

```
1 def calculate_ber(original: np.ndarray, received: np.ndarray)
  -> float:
```

8 Теоретические кривые BER

Описание: Теоретические кривые BER показывают, какой BER ожидается при различных значениях SNR.

Зачем нужен: Для сравнения с экспериментальными результатами.

Теоретическое описание: Теоретические кривые BER основаны на математических моделях, которые предсказывают BER в зависимости от SNR (Signal-to-Noise Ratio). Эти кривые позволяют сравнить теоретические

ожидания с реальными результатами.

Формула (SISO):

$$\text{BER}_{\text{SISO}} = 0.5 \left(1 - \sqrt{\frac{\text{SNR}}{1 + \text{SNR}}} \right)$$

Объяснение: Формула для SISO (Single Input Single Output) системы предсказывает BER в зависимости от SNR. Здесь SNR — это отношение сигнал/шум. Формула учитывает влияние шума на передачу сигнала и предсказывает долю ошибочных битов.

Формула (Alamouti):

$$p_{\text{Alamouti}} = \frac{1}{2} - \frac{1}{2} \left(1 + \frac{2}{\text{SNR}} \right)^{-1/2}$$

$$\text{BER}_{\text{Alamouti}} = p_{\text{Alamouti}}^2 [1 + 2(1 - p_{\text{Alamouti}})]$$

Объяснение: Формула для системы с кодированием Аламути предсказывает BER в зависимости от SNR. Здесь SNR — это отношение сигнал/шум. Формула учитывает ортогональные свойства кодов Аламути и предсказывает долю ошибочных битов. **Вспомогательный материал:**

- **C++:** Использование `std::pow`, `std::sqrt`.
- **Python:** Использование `numpy.power`, `numpy.sqrt`.

Сигнатура функции (SISO):

```
1 double theoretical_ber_asiso(double SNRdB);
```

```
1 def theoretical_ber_asiso(SNRdB: float) -> float:
```

Сигнатура функции (MIMO ZF):

```
1 double theoretical_ber_mimo_zf(double SNRdB);
```

```
1 def theoretical_ber_mimo_zf(SNRdB: float) -> float:
```

Сигнатура функции (Alamouti):

```
1 double theoretical_ber_alamouti(double SNRdB);
```

```
1 def theoretical_ber_alamouti(SNRdB: float) -> float:
```