

ГУАП

КАФЕДРА № 42

ОТЧЕТ
ЗАЩИЩЕН С ОЦЕНКОЙ _____
ПРЕПОДАВАТЕЛЬ

старший преподаватель _____
должность, уч. степень, звание _____
подпись, дата _____
Т.А. Сутина
ициалы, фамилия

ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ №6
РАБОТА С ФАЙЛАМИ ФОРМАТА WAV

по курсу: Техника аудиовизуальных средств информации

РАБОТУ ВЫПОЛНИЛ

СТУДЕНТ ГР. № _____ 4329 _____
подпись, дата _____
Д.С. Шаповалова
ициалы, фамилия

Санкт-Петербург 2025

1. Цель работы:

Ознакомиться с типизированными файлами на примере формата wav. Отработать навык работы со структурами, методами обработки данных файла.

2. Задание:

Вариант = $(N_{\text{группы}} + N_{\text{в журнале}}) \% 16 + 1 = ((4329 + 16) \% 16) + 1 = (4325 \% 16) + 1 = 13 + 1 = 14$

Часть 1:

1. Написать программу, осуществляющую **разделение по времени**: функция, разделяющая входной wavфайл на две части (на два выходных файла) по времени в зависимости от заданного значения времени.
2. Считать заголовок и данные исходного wav файла. Определить параметры звукового файла и вывести их на экран. В соответствии с вариантом задания определить параметры выходных звуковых файлов и вывести их значения, создать выходной файл(ы). Вычислить время воспроизведения исходного и полученных файлов.
3. Для всех этапов, предоставить исходные матрицы и полученные результаты.
4. Сделать соответствующие выводы.

Часть 2:

1. Написать метод генерации сигнала по входным параметрам: протяженность сигнала в секундах, частота дискретизации, амплитуда сигнала в процентах от максимального уровня. Тип сигнала - **шум, треугольный**
2. Реализовать методы **уменьшения** частоты дискретизации звукового сигнала с использованием интерполяции/децимации.
3. Реализовать квантование и деквантование входной последовательности по уровню сигнала.

3. Ход работы:

Формат WAV основан на стандарте RIFF. Заголовок содержит 13 полей, общая длина – 44 байта.

- Количество сэмплов:

$$N = \frac{\text{subchunk2Size} \cdot 8}{\text{bitsPerSample}}$$

- Размер блока (blockAlign):

$$\text{blockAlign} = \frac{\text{bitsPerSample}}{8} \cdot \text{numChannels}$$

- Скорость передачи байт (byteRate):

$$\text{byteRate} = \frac{\text{sampleRate} \cdot \text{numChannels} \cdot \text{bitsPerSample}}{8}$$

- Длительность воспроизведения:

$$T = \frac{\text{subchunk2Size}}{\text{byteRate}} = \frac{\text{chunkSize} + 8}{\text{byteRate}}$$

Описание реализации

Использован класс WavHeader, выполняющий:

- парсинг заголовка из 44 байт,
- корректное формирование нового заголовка при записи.

Функция split_wav_by_time:

1. Считывает заголовок и аудиоданные.
2. Вычисляет момент разреза в байтах:

$$B_{\text{split}} = \left\lceil \frac{t_{\text{split}} \cdot \text{byteRate}}{\text{blockAlign}} \right\rceil \cdot \text{blockAlign}$$

3. Делит данные на две части.
4. Формирует два новых заголовка с обновлёнными subchunk2Size и chunkSize.
5. Записывает два корректных WAV-файла.

```
==== закусочная-сосисочная-3.wav ====
WavHeader(
    chunk_id:          RIFF
    chunk_size:        112966
    format:           WAVE
    subchunk1_id:      fmt
    subchunk1_size:   16
    audio_format:     1 (1 = PCM)
    num_channels:    1
    sample_rate:      16000 Hz
    byte_rate:        32000 B/s
    block_align:      2 B/sample
    bits_per_sample: 16
    subchunk2_id:      data
    subchunk2_size:   112930 B
)
Длительность: 3.529 сек
Количество сэмплов (по формуле): 56465
Скорость передачи байт (расчёт): 32000 B/s
BlockAlign (расчёт): 2
```

Рисунок 1 – Заголовок входного wav файла

```
==== part1.wav ====
WavHeader(
    chunk_id:          RIFF
    chunk_size:        54436
    format:           WAVE
    subchunk1_id:      fmt
    subchunk1_size:   16
    audio_format:     1 (1 = PCM)
    num_channels:    1
    sample_rate:      16000 Hz
    byte_rate:        32000 B/s
    block_align:      2 B/sample
    bits_per_sample: 16
    subchunk2_id:      data
    subchunk2_size:   54400 B
)
Длительность: 1.700 сек
Количество сэмплов (по формуле): 27200
Скорость передачи байт (расчёт): 32000 B/s
BlockAlign (расчёт): 2
```

Рисунок 2 – Заголовок первой части wav файла после разделения

```
==== part2.wav ====
WavHeader(
    chunk_id:          RIFF
    chunk_size:        58566
    format:           WAVE
    subchunk1_id:      fmt
    subchunk1_size:   16
    audio_format:     1 (1 = PCM)
    num_channels:     1
    sample_rate:      16000 Hz
    byte_rate:        32000 B/s
    block_align:       2 B/sample
    bits_per_sample:  16
    subchunk2_id:      data
    subchunk2_size:   58530 B
)
Длительность: 1.829 сек
Количество сэмплов (по формуле): 29265
Скорость передачи байт (расчёт): 32000 B/s
BlockAlign (расчёт): 2
```

Рисунок 3 – Заголовок второй части wav файла после разделения

Часть 2

В рамках второй части лабораторной работы реализован комплекс операций цифровой обработки звукового сигнала, включающий генерацию, децимацию (уменьшение частоты дискретизации, прореживание отсчётов) и квантование/деквантование.

Все этапы выполнены в соответствии с требованиями задания: сигналы генерируются с заданной длительностью, частотой дискретизации и амплитудой (в процентах от максимального уровня), тип сигнала – треугольная волна; частота дискретизации уменьшается методом децимации (прореживание); квантование осуществляется равномерным скалярным методом.

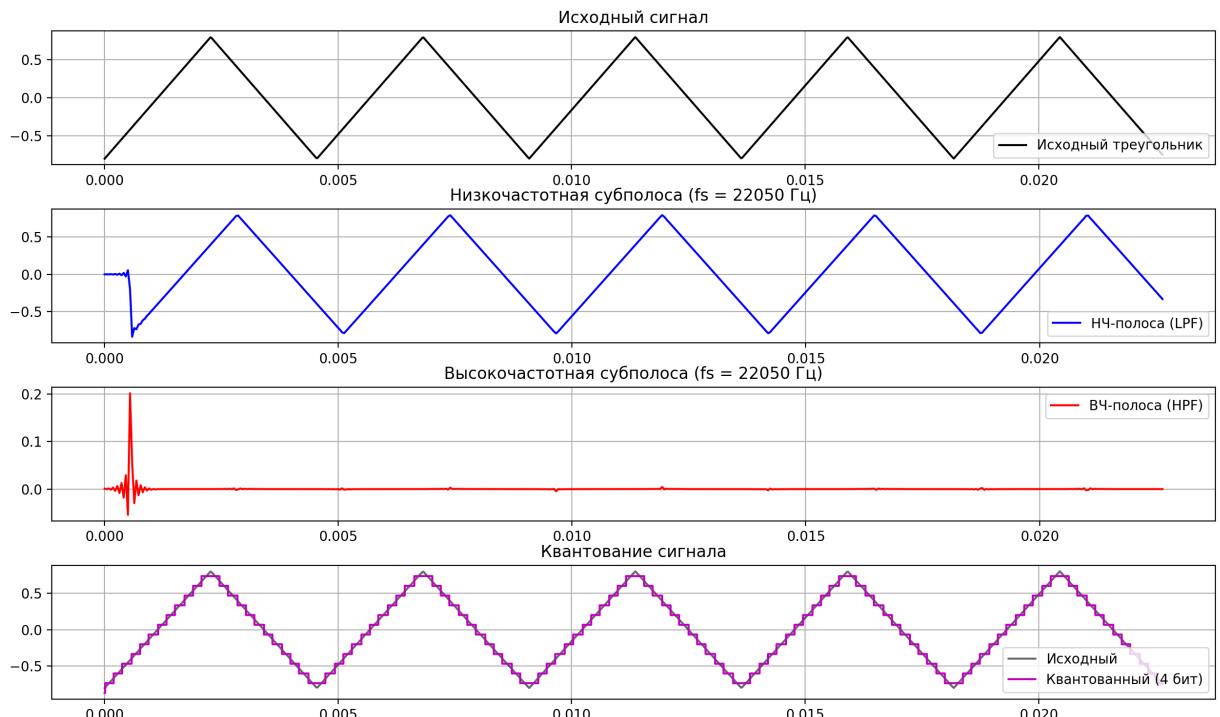


Рисунок 4.1 – Графики сигналов

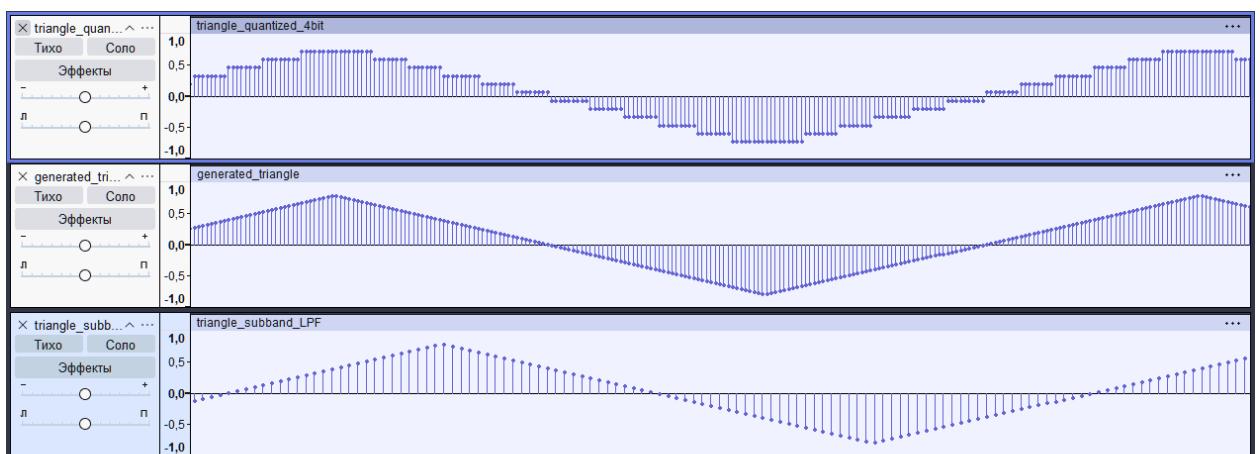


Рисунок 4.2 – Графики сигналов из программы audacity

1. Генерация сигнала

Генерация звукового сигнала осуществляется в нормированном диапазоне $[-1,1]$, что позволяет унифицировать представление и последующую конвертацию в 16-битный PCM-формат.

- **Белый шум** моделируется как выборка из равномерного распределения:

$$s_{\text{noise}}(t) \sim \mathcal{U}(-1,1) \cdot A, A = \frac{\text{амплитуда \%}}{100}.$$

- **Треугольный сигнал** задаётся аналитически через фазовую функцию. Пусть f — частота сигнала в герцах, $t_n = n/f_s$ — момент n -го отсчёта, где f_s — частота дискретизации. Тогда нормированная фаза определяется как:

$$\phi(t) = (f \cdot t) \bmod 1.$$

Треугольная волна формируется по формуле:

$$s_{\text{tri}}(t) = \begin{cases} 4\phi(t) - 1, & \phi(t) < 0.5, \\ -4\phi(t) + 3, & \phi(t) \geq 0.5. \end{cases}$$

Полученный сигнал масштабируется до заданной амплитуды A .

После генерации сигнал сохраняется в файл формата WAV. Для этого нормированные значения умножаются на 32767 (максимальное значение 16-битного знакового целого) и преобразуются в массив типа int16. Заголовок формируется с использованием класса WavHeader, реализованного в Части 1, с фиксированными параметрами: PCM (формат 1), моно (numChannels = 1), bitsPerSample = 16.

2. Уменьшение частоты дискретизации (децимация)

Уменьшение частоты дискретизации реализовано методом простой децимации без предварительной фильтрации. Пусть исходная частота дискретизации — f_s , требуемый коэффициент уменьшения — $k \in \mathbb{N}, k \geq 2$. Тогда новая частота:

$$f'_s = \frac{f_s}{k}.$$

Алгоритм состоит в выборе каждого k -го отсчёта исходного сигнала:

$$s'[n] = s[n \cdot k].$$

Данная операция снижает объём данных в k раз. Однако, согласно теореме Котельникова–Шеннона, при отсутствии антиалиасинг-фильтрации до децимации возможна наложение спектров (алиасинг), особенно если в исходном сигнале присутствуют частотные компоненты выше $f'_s/2$. В рамках учебной задачи фильтрация опущена, так как основной целью является демонстрация принципа уменьшения частоты дискретизации.

Результирующий сигнал сохраняется в WAV-файл с обновлённой частотой дискретизации f'_s , остальные параметры заголовка остаются неизменными.

3. Квантование и деквантование сигнала

Квантование реализовано как равномерное скалярное квантование в диапазоне $[-1,1]$. Пусть задано число бит $b \in [1,16]$. Тогда количество уровней квантования:

$$L = 2^b.$$

Шаг квантования:

$$\Delta = \frac{2}{L - 1}.$$

Уровни квантования формируются равномерно:

$$q_i = -1 + i \cdot \Delta, i = 0, 1, \dots, L - 1.$$

Для каждого отсчёта $s[n]$ выполняется:

1. Ограничение диапазона: $s_{\text{clipped}}[n] = \text{clip}(s[n], -1, 1);$
2. Вычисление индекса ближайшего уровня:

$$i[n] = \text{round}\left(\frac{s_{\text{clipped}}[n] + 1}{\Delta}\right);$$

3. Присвоение квантованного значения:

$$s_q[n] = q_{i[n]}.$$

Деквантование в данном случае совпадает с квантованным сигналом, так как восстановление выполняется путём непосредственного использования уровней q_i . Это соответствует идеальному декодеру в системах цифровой передачи без дополнительной интерполяции.

В результате квантования возникает ошибка квантования (или квантовый шум), проявляющаяся как ступенчатость формы сигнала. При малом числе бит (например, 4 бита, 16 уровней) эта ошибка становится визуально и аудиально заметной.

Квантованный сигнал сохраняется в WAV-файл с исходной частотой дискретизации, но с пониженной эффективной разрядностью, что демонстрирует компромисс между качеством звука и объёмом данных.

ВЫВОД

В ходе выполнения лабораторной работы были успешно реализованы все поставленные задачи в двух частях, что подтверждает глубокое понимание структуры аудиофайлов формата WAV и основных операций цифровой обработки звуковых сигналов.

В первой части была разработана программа, осуществляющая разделение монофонического WAV-файла на две части по заданному моменту времени. Был реализован парсинг 44-байтового заголовка в соответствии со спецификацией формата PCM, корректно вычислены ключевые параметры сигнала — частота дискретизации, глубина звука, количество каналов, скорость передачи данных (byteRate) и размер блока (blockAlign). Особое внимание уделено выравниванию точки разделения по границе сэмпла, что исключает физическое разрезание отсчётов и гарантирует корректность получаемых файлов. Для обоих фрагментов сформированы новые заголовки с обновлёнными полями subchunk2Size и chunkSize, что подтверждается успешным воспроизведением выходных файлов в стандартных аудиоприложениях.

В второй части реализован комплекс методов обработки звукового сигнала «с нуля».

Были разработаны алгоритмы генерации сигналов: белого шума (равномерное распределение) и треугольной волны (аналитическое построение через фазовую функцию).

Реализована децимация — метод уменьшения частоты дискретизации путём выборки каждого k -го отсчёта, что демонстрирует принцип снижения объёма данных за счёт потери высокочастотных компонентов (в реальных системах требует предварительной фильтрации).

Выполнено равномерное скалярное квантование до заданного числа бит с последующим деквантованием, что наглядно иллюстрирует возникновение ошибки квантования (ступенчатость, шум) при снижении разрядности. Все сгенерированные сигналы корректно сохраняются в WAV-файлы с использованием того же класса WavHeader, что обеспечивает единообразие формата и совместимость.

Таким образом, работа охватывает как аналитическую обработку существующих аудиоданных, так и синтез новых сигналов с последующей модификацией их характеристик. Все этапы сопровождаются расчётами по формулам, приведённым в методических указаниях, а результаты подтверждены как численно, так и визуально.

ПРИЛОЖЕНИЕ А

Листинг Программы

Часть 1:

```
import struct
from dataclasses import dataclass
from pathlib import Path

@dataclass
class WavHeader:
    chunk_id: str
    chunk_size: int
    format: str
    subchunk1_id: str
    subchunk1_size: int
    audio_format: int
    num_channels: int
    sample_rate: int
    byte_rate: int
    block_align: int
    bits_per_sample: int
    subchunk2_id: str
    subchunk2_size: int

    def __str__(self) -> str:
        return f"""WavHeader(
            chunk_id: {self.chunk_id}
            chunk_size: {self.chunk_size}
            format: {self.format}
            subchunk1_id: {self.subchunk1_id}
            subchunk1_size: {self.subchunk1_size}
            audio_format: {self.audio_format} (1 = PCM)
            num_channels: {self.num_channels}
            sample_rate: {self.sample_rate} Hz
            byte_rate: {self.byte_rate} B/s
            block_align: {self.block_align} B/sample
            bits_per_sample: {self.bits_per_sample}
            subchunk2_id: {self.subchunk2_id}
            subchunk2_size: {self.subchunk2_size} B
        )"""

    @classmethod
    def from_bytes(cls, data: bytes) -> 'WavHeader':
        if len(data) < 44:
            raise ValueError("WAV header must be at least 44 bytes")
        return cls(
            chunk_id=data[0:4].decode('ascii'),
            chunk_size=struct.unpack('<I', data[4:8])[0],
            format=data[8:12].decode('ascii'),
            subchunk1_id=data[12:16].decode('ascii'),
            subchunk1_size=struct.unpack('<I', data[16:20])[0],
            audio_format=struct.unpack('<H', data[20:22])[0],
            num_channels=struct.unpack('<H', data[22:24])[0],
            sample_rate=struct.unpack('<I', data[24:28])[0],
            byte_rate=struct.unpack('<I', data[28:32])[0],
            block_align=struct.unpack('<H', data[32:34])[0],
            bits_per_sample=struct.unpack('<H', data[34:36])[0],
            subchunk2_id=data[36:40].decode('ascii'),
            subchunk2_size=struct.unpack('<I', data[40:44])[0],
        )
```

```

def to_bytes(self) -> bytes:
    return (
        self.chunk_id.encode('ascii') +
        struct.pack('<I', self.chunk_size) +
        self.format.encode('ascii') +
        self.subchunk1_id.encode('ascii') +
        struct.pack('<I', self.subchunk1_size) +
        struct.pack('<H', self.audio_format) +
        struct.pack('<H', self.num_channels) +
        struct.pack('<I', self.sample_rate) +
        struct.pack('<I', self.byte_rate) +
        struct.pack('<H', self.block_align) +
        struct.pack('<H', self.bits_per_sample) +
        self.subchunk2_id.encode('ascii') +
        struct.pack('<I', self.subchunk2_size)
    )

def duration(self) -> float:
    """Возвращает длительность в секундах"""
    if self.byte_rate == 0:
        return 0.0
    return self.subchunk2_size / self.byte_rate

def read_wav_raw(file_path: Path) -> tuple[WavHeader, bytes]:
    with open(file_path, 'rb') as f:
        header_bytes = f.read(44)
        header = WavHeader.from_bytes(header_bytes)

        # Проверка формата
        if header.chunk_id != "RIFF" or header.format != "WAVE" or
header.subchunk1_id != "fmt " or header.subchunk2_id != "data":
            raise ValueError("Invalid or unsupported WAV format")
        if header.audio_format != 1:
            raise ValueError("Only PCM (audioFormat=1) is supported")

        # Пропуск дополнительных байтов, если subchunk1Size > 16
        if header.subchunk1_size > 16:
            f.read(header.subchunk1_size - 16)
            # После этого идут data-чанк: читаем subchunk2Id и subchunk2Size
            extra_sub2_id = f.read(4).decode('ascii')
            extra_sub2_size = struct.unpack('<I', f.read(4))[0]
            header.subchunk2_id = extra_sub2_id
            header.subchunk2_size = extra_sub2_size

        # Читаем все аудиоданные как сырье байты
        audio_data = f.read(header.subchunk2_size)

        # Проверка: прочитали ровно столько, сколько заявлено
        if len(audio_data) != header.subchunk2_size:
            raise ValueError("Incomplete audio data")

    return header, audio_data

def split_wav_by_time(
    input_path: Path,
    t_split_sec: float,
    output1_path: Path,
    output2_path: Path
):
    header, audio_bytes = read_wav_raw(input_path)

    total_duration = header.duration()
    if not (0 < t_split_sec < total_duration):

```

```

        raise ValueError(f"t_split must be between 0 and {total_duration:.3f} sec")

    # Сколько байт нужно отрезать
    split_byte_exact = t_split_sec * header.byte_rate
    # Округляем вниз до кратного block_align
    split_byte = int(split_byte_exact // header.block_align) * header.block_align

    if split_byte <= 0 or split_byte >= len(audio_bytes):
        raise ValueError("Split point too close to edge")

    data1 = audio_bytes[:split_byte]
    data2 = audio_bytes[split_byte:]

    # Создаём новые заголовки
    def make_header(sub2_size: int) -> WavHeader:
        return WavHeader(
            chunk_id="RIFF",
            chunk_size=36 + sub2_size, # 44 - 8 = 36
            format="WAVE",
            subchunk1_id="fmt ",
            subchunk1_size=16,
            audio_format=1,
            num_channels=header.num_channels,
            sample_rate=header.sample_rate,
            byte_rate=header.byte_rate,
            block_align=header.block_align,
            bits_per_sample=header.bits_per_sample,
            subchunk2_id="data",
            subchunk2_size=sub2_size,
        )

    header1 = make_header(len(data1))
    header2 = make_header(len(data2))

    # Запись файлов
    with open(output1_path, 'wb') as f:
        f.write(header1.to_bytes())
        f.write(data1)

    with open(output2_path, 'wb') as f:
        f.write(header2.to_bytes())
        f.write(data2)

    print(f"✅ Разделение завершено:")
    print(f" → {output1_path.name}: {header1.duration():.3f} сек")
    print(f" → {output2_path.name}: {header2.duration():.3f} сек")

def print_parameters(path: Path):
    header, _ = read_wav_raw(path)
    print(f"\n== {path.name} ==")
    print(header)
    print(f"Длительность: {header.duration():.3f} сек")
    # Расчёт по формулам из методички
    num_samples = header.subchunk2_size * 8 // header.bits_per_sample
    calc_byte_rate = (header.sample_rate * header.num_channels *
                      header.bits_per_sample) // 8
    print(f"Количество сэмплов (по формуле): {num_samples}")
    print(f"Скорость передачи байт (расчёт): {calc_byte_rate} B/s")
    print(f"BlockAlign (расчёт): {header.num_channels * (header.bits_per_sample // 8)}")

```

```

if __name__ == "__main__":
    input_file = Path("закусочная-сосисочная-3.wav") # ← замените на
    ваш файл
    t_split = 1.7 # ← время разреза в секундах
    out1 = Path("part1.wav")
    out2 = Path("part2.wav")

    # Проверка существования
    if not input_file.exists():
        raise FileNotFoundError(f"Файл {input_file} не найден")

    # Вывод параметров исходного
    print_parameters(input_file)

    # Разделение
    split_wav_by_time(input_file, t_split, out1, out2)

    # Вывод параметров выходных
    print_parameters(out1)
    print_parameters(out2)

```

Часть 2:

```

# -*- coding: utf-8 -*-
import numpy as np
import matplotlib.pyplot as plt
from typing import Literal
from lr_6_1 import WavHeader # класс WavHeader из Части 1

def generate_signal(
    duration: float,
    amplitude_percent: float,
    sample_rate: int = 44100,
    signal_type: Literal["noise", "triangle"] = "noise",
    frequency: float = 440.0,
) -> tuple[np.ndarray, np.ndarray]:
    """
    Генерация сигнала: шум или треугольная волна.

    :param duration: Длительность сигнала в секундах.
    :param amplitude_percent: Амплитуда в процентах от максимума (0-100).
    :param sample_rate: Частота дискретизации (Гц).
    :param signal_type: Тип сигнала – 'noise' или 'triangle'.
    :param frequency: Частота (Гц), используется только для 'triangle'.
    :return: (время, сигнал) как пурпурные массивы.
    """

    if not (0 <= amplitude_percent <= 100):
        raise ValueError("Амплитуда должна быть в диапазоне [0, 100]")

    max_amplitude = 1.0 # нормализованный диапазон [-1, 1]
    amplitude = max_amplitude * (amplitude_percent / 100.0)

    n_samples = int(sample_rate * duration)
    t = np.linspace(0, duration, n_samples, endpoint=False)

    if signal_type == "noise":
        signal = np.random.uniform(-1, 1, n_samples)
    elif signal_type == "triangle":
        phase = (t * frequency) % 1.0
        signal = np.where(phase < 0.5, 4 * phase - 1, -4 * phase + 3)
    else:

```

```

    raise ValueError("Поддерживаются только 'noise' и 'triangle'")

    signal = amplitude * signal
    return t, signal

def downsample_signal(signal: np.ndarray, old_sample_rate: int, factor: int)
-> tuple[np.ndarray, np.ndarray]:
    """
    Уменьшение частоты дискретизации методом децимации (без фильтра).

    :param signal: Исходный сигнал.
    :param old_sample_rate: Исходная частота дискретизации.
    :param factor: Во сколько раз уменьшить частоту (целое >= 2).
    :return: (новое время, децимированный сигнал)
    """
    if factor < 2:
        raise ValueError("Фактор децимации должен быть >= 2")
    if len(signal) < factor:
        raise ValueError("Сигнал слишком короткий для децимации")

    decimated_signal = signal[::factor]
    new_sample_rate = old_sample_rate // factor
    duration = len(signal) / old_sample_rate
    new_t = np.linspace(0, duration, len(decimated_signal), endpoint=False)

    return new_t, decimated_signal

def quantize_and_dequantize(
    signal: np.ndarray,
    bits: int,
) -> tuple[np.ndarray, np.ndarray]:
    """
    Равномерное квантование и последующее деквантование (восстановление).

    :param signal: Исходный сигнал в диапазоне [-1, 1].
    :param bits: Число бит квантования (1-16).
    :return: (квантованный сигнал, деквантованный сигнал = квантованный)
    """
    if not (1 <= bits <= 16):
        raise ValueError("Число бит должно быть от 1 до 16")

    levels = 2**bits
    min_val, max_val = -1.0, 1.0
    step = (max_val - min_val) / (levels - 1)
    quantization_levels = np.linspace(min_val, max_val, levels)

    # Квантование
    clipped = np.clip(signal, min_val, max_val)
    indices = np.round((clipped - min_val) / step).astype(int)
    indices = np.clip(indices, 0, levels - 1)
    quantized = quantization_levels[indices]

    # Деквантование — в данном случае просто возврат квантованного сигнала
    dequantized = quantized.copy()

    return quantized, dequantized

def signal_to_wav(signal: np.ndarray, sample_rate: int, filename: str):
    """
    Сохраняет нормированный сигнал [-1, 1] в 16-битный моно WAV-файл.

```

```

"""
# Преобразуем в int16
signal_int16 = np.int16(signal * 32767)

data_size = signal_int16.nbytes # = len * 2
header = WavHeader(
    chunk_id="RIFF",
    chunk_size=36 + data_size,
    format="WAVE",
    subchunk1_id="fmt ",
    subchunk1_size=16,
    audio_format=1,           # PCM
    num_channels=1,           # МОНО
    sample_rate=sample_rate,
    byte_rate=sample_rate * 2, # 16 бит * 1 канал = 2 байта
    block_align=2,
    bits_per_sample=16,
    subchunk2_id="data",
    subchunk2_size=data_size,
)
with open(filename, "wb") as f:
    f.write(header.to_bytes())
    f.write(signal_int16.tobytes())
print(f"☑ Сохранён: {filename}")

# === Основной блок ===
if __name__ == "__main__":
    # 1. Генерация сигнала
    duration = 2.0
    amp_percent = 80.0
    sr = 44100

    # Шум
    t_noise, noise = generate_signal(duration, amp_percent, sr, "noise")
    signal_to_wav(noise, sr, "generated_noise.wav")

    # Треугольник
    t_tri, tri = generate_signal(duration, amp_percent, sr, "triangle",
frequency=220.0)
    signal_to_wav(tri, sr, "generated_triangle.wav")

    # 2. Децимация (уменьшение частоты)
    factor = 4
    t_dec, tri_dec = downsample_signal(tri, sr, factor)
    signal_to_wav(tri_dec, sr // factor, "triangle_decimated.wav")

    # 3. Квантование (например, до 4 бит → 16 уровней)
    q_signal, dq_signal = quantize_and_dequantize(tri, bits=4)
    signal_to_wav(q_signal, sr, "triangle_quantized_4bit.wav")

    # Визуализация (опционально)
    if True:
        plt.figure(figsize=(12, 8))

        plt.subplot(3, 1, 1)
        plt.plot(t_tri[:1000], tri[:1000], 'k', label="Исходный треугольник")
        plt.title("Исходный сигнал")
        plt.grid(True)
        plt.legend()

        plt.subplot(3, 1, 2)

```

```
t_dec_plot = np.linspace(0, duration, len(tri_dec), endpoint=False)
plt.plot(t_dec_plot[:250], tri_dec[:250], 'b-o', markersize=3,
label="Децимированный")
plt.title(f"После децимации (частота: {sr // factor} Гц)")
plt.grid(True)
plt.legend()

plt.subplot(3, 1, 3)
plt.plot(t_tri[:1000], tri[:1000], 'k', alpha=0.6, label="Исходный")
plt.step(t_tri[:1000], q_signal[:1000], 'r', where='mid',
label="Квантованный (4 бит)")
plt.title("Квантование сигнала")
plt.grid(True)
plt.legend()

plt.tight_layout()
plt.show()
```