МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«СЕВАСТОПОЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Институт радиоэлектроники и интеллектуальных технических систем
Кафедра «Информатика и управление в технических системах»

КУРСОВАЯ РАБОТА

по дисциплине «Обработка данных в автоматизированных системах»
на тему
«Прикладная обработка данных»

Выполнил:
ст. гр. УТС/б-22-01-о
Анненков В. Д.

Проверил:
к.т.н., доцент кафедры ИУТС
Альчаков В. В.

Севастополь – 2024

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«СЕВАСТОПОЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Институт: Институт радиоэлектроники и интеллектуальных технических систем
(наименование института полностью)

Кафедра: Информатика и управление в технических системах
(наименование кафедры полностью)

27.03.04 Управление в технических системах
(код и наименование направления подготовки/специальности)

Интеллектуальные робототехнические системы
(наименование профиля/специализации)

Курс: 3 Группа: УТС/б22-1-о Семестр: 3

З А Д А Н И Е
НА КУРСОВУЮ РАБОТУ

Обучающегося Анненкова Владимира Дмитриевича
(фамилия, имя, отчество)

1. Тема работы: Прикладная обработка данных. Сравнение нейросетевого алгоритма на основе автоэнкодера для задачи фильтрации зашумленного сигнала с традиционными алгоритмами.

2. Срок сдачи обучающимся законченной работы _____

3. Исходные/входные данные к работе_____
_____
_____
_____
_____

4. Содержание пояснительной записки (перечень вопросов, подлежащих разработке)
_____
_____
_____
_____
_____
_____

5. Перечень графического материала (с точным указанием обязательных чертежей) _____
_____
_____

6. Дата выдачи задания_____

КАЛЕНДАРНЫЙ ПЛАН

№ п/п	Название этапов выполнения курсовой работы	Срок выполнения этапов работы	Примечание

Обучающийся Анненков В. Д
(фамилия и инициалы) (подпись)

Руководитель курсовой работы _____ _____
(фамилия, инициалы, должность) (подпись)

«_____» декабрь 2024 г.

Оглавление

ВВЕДЕНИЕ

В современном мире, где цифровая обработка сигналов играет ключевую роль в самых разных областях – от телекоммуникаций и медицины до аудио- и видеообработки – задача эффективного выделения полезного сигнала из шума приобретает первостепенное значение. Шум, будь то непрерывный фоновый шум или кратковременные импульсные помехи, неизбежно искажает полезную информацию, снижая качество данных и затрудняя их интерпретацию. Традиционные методы фильтрации имеют ряд ограничений, особенно в условиях сложной шумовой обстановки и нелинейных искажений.

В последние годы, с развитием технологий машинного обучения и, в частности, нейронных сетей, открываются новые перспективы в области обработки сигналов. Нейросетевые модели, способные к обучению сложным нелинейным зависимостям, предоставляют мощный инструмент для фильтрации сигналов, адаптируясь к различным типам шума и обеспечивая более точное восстановление полезного сигнала.

Актуальность темы

Актуальность данной работы обусловлена возрастающей потребностью в надежных и эффективных методах фильтрации сигналов. Использование нейронных сетей в данной области представляет собой перспективное направление, способное превзойти традиционные методы по точности и адаптивности. Исследование и сравнение эффективности нейросетевых подходов с классическими алгоритмами позволит лучше понять преимущества и ограничения каждого метода, а также определить области их наиболее эффективного применения.

Цель работы

Основной целью данной курсовой работы является исследование и сравнение эффективности нейросетевого подхода с традиционными алгоритмами фильтрации (скользящее среднее, фильтр Винера и фильтр низких частот) применительно к задачам фильтрации сигналов, зашумленных непрерывным и импульсным шумом.

Задачи работы

1. Генерация зашумленных сигналов: создать набор тестовых сигналов, зашумленных как непрерывным (например, гауссовским) шумом, так и импульсным (например, солевой и перцовый) шумом, используя библиотеку $NumPy$ для генерации массивов и случайных чисел.
2. Реализация традиционных алгоритмов: разработать программные реализации алгоритмов скользящего среднего, фильтра Винера и фильтра низких частот, используя библиотеки $numpy,\ matplotlib$.
3. Разработка нейросетевой модели: создать нейросетевую модель (например, рекуррентную или сверточную) для фильтрации сигналов, используя библиотеку $TensorFlow$ или $PyTorch$.
4. Обучение нейросети: обучить разработанную нейросетевую модель на наборе зашумленных и чистых сигналов с использованием алгоритмов оптимизации, предоставляемых $TensorFlow$ или $PyTorch$.
5. Оценка качества фильтрации: оценить качество фильтрации каждого метода (традиционного и нейросетевого) с использованием метрик, таких как среднеквадратичная ошибка (MSE), отношение сигнал/шум (SNR) и структурное сходство (SSIM), используя библиотеки $NumPy$ и $scikit-learn$ для вычисления этих метрик.
6. Сравнение результатов: провести сравнительный анализ эффективности нейросетевого подхода и традиционных алгоритмов фильтрации на основе полученных оценок качества, используя библиотеку $Matplotlib$ и $Seaborn$ для визуализации результатов.

7. Формулирование выводов: сформулировать выводы о применимости и эффективности каждого из рассмотренных методов фильтрации в различных условиях и дать рекомендации по их применению.

1 Моделирование полигармонического сигнала с шумом

1.1 Теоретические сведения

Полигармонический сигнал является удобной моделью для представления многих реальных сложных сигналов. Он состоит из нескольких гармонических компонентов с разными частотами, амплитудами и фазами. Применений полигармоническому сигналу множество, например использование в радиолокации и обработки сигналов. Полигармонический сигнал описывается следующим образом:

$$x(t) = A \sum_{i=1}^{N} sin(2\pi f t) \tag{1.1}$$

$x(t)$ – полученный полигармонический сигнал, A – амплитуда, f – значения частот, $t -$ время, $M -$ количество гармонических составляющих сигнала.

В обработке сигналов шум — это общий термин, обозначающий нежелательные (и, в общем случае, неизвестные) модификации, которым сигнал может подвергаться во время захвата, хранения, передачи, обработки или преобразования. Шум в полигармонических сигналах представляет собой случайные колебания амплитуды и фазы компонентов сигнала.

Это означает, что на каждой гармонике в полигармоническом сигнале происходят случайные изменения, что приводит к добавлению шума. Присутствие шума в полигармонических сигналах может оказывать влияние на процессы обработки и анализа этих сигналов. Он может искажать сигналы, снижать качество передачи информации и усложнять его извлечение. Наложение шума описывается следующим выражением:

$$x_\eta(t) = x(t) + \eta(t), \tag{1.2}$$

где $x_\eta(t) -$ значение зашумленного полигармонического сигнала,

$\eta(t)$ – зашумленный сигнал, $x(t)$ – исходный полигармонический сигнал.

Аддитивный шум представляет собой случайные сигналы, которые добавляются к основному сигналу или сигналу интереса.

Выделяют несколько видов аддитивных шумов, по их отношению к исходному сигналу:

Непрерывный шум имеет постоянный характер и существует в течение длительного времени. Он часто характеризуется как фоновый шум и может быть равномерно распределён по частотному спектру (например, белый шум) или иметь определённый спектральный диапазон (розовый шум, коричневый шум).

Прерывистый шум возникает как последовательность коротких «вспышек» или интервалов шума, которые могут иметь нерегулярный характер. Размеры интервалов зачастую случайны, что затрудняет задачу фильтрации. Часто связан с внешними источниками или техническими неисправностями.

Импульсный шум состоит из серии коротких высокоамплитудных импульсов, которые имеют ограниченную длительность, но значительную мощность. Его частотный спектр обычно широк. Величина амплитуды шума обычно равно или превосходит амплитуды исследуемого сигнала.

Шум с узкополосным спектром — это вид шума ограничен узким диапазоном частот. Он может быть вызван как внутренними особенностями системы (регенерация, колебания), так и внешними гармоническими помехами.

1.2 Практическая реализация

Для генерации и визуализации полигармонического сигнала будем использовать средства языка *Python*, с библиотеками *Numpy* и *Matplotlib*. Сначала импортируем библиотеки и установим значения параметров настройки отображения графиков:

```
import numpy as np
import matplotlib.pyplot as plt
```

```
# Настройка представления
get_ipython().run_line_magic('matplotlib', 'inline')
get_ipython().run_line_magic('config', "InlineBackend.figure_format = 'svg'")
```

Теперь зададим частоты отдельных гармоник, а также амплитуду сигнала нашего сигнала:

```
A = 5
f1 = 20
f2 = 40
f3 = 60
f4 = 80
f5 = 100
N = 1024
T = 0.0005
i = np.arange(N)
f = np.array([f1, f2, f3, f4, f5])
```

Сгенерируем итоговый сигнал, сложив его отдельные гармоники:

```
plt.figure(figsize=(18, 6), dpi=300)
for k in np.arange(f.size):
        x = np.sin(2*np.pi*f[k]*T*i[:150])
        plt.plot(i[:150]*T, x)
        plt.title('Сигнал(t)', fontsize=14)
        plt.xlabel('t [сек]', fontsize=12)
        plt.ylabel('A(t) [В]', fontsize=12)
        plt.grid(True)
plt.savefig(r'fig_w.png')
plt.show()
```

Отрисовка графика состоит из следующих этапов:

1) $plt.figure(figsize = (18,6), dpi = 300)$: Создает фигуру(окно) для отрисовки графика, задает ей размер 18 на 6 дюймов и разрешение 300
2) $for\ k\ in\ np.arange(f.size)$:: Цикл, в котором для каждой гармоники рисуется отдельный график

3) $x = np.sin(2 * np.pi * f[k] * T * i[:150])$: Считает значения гармоники для первых 150 отсчетов
4) $plt.plot(i[:150] * T, x)$: Рисует график гармоники по первым 150 отсчетам, по оси x - время в секундах, по y - значение сигнала

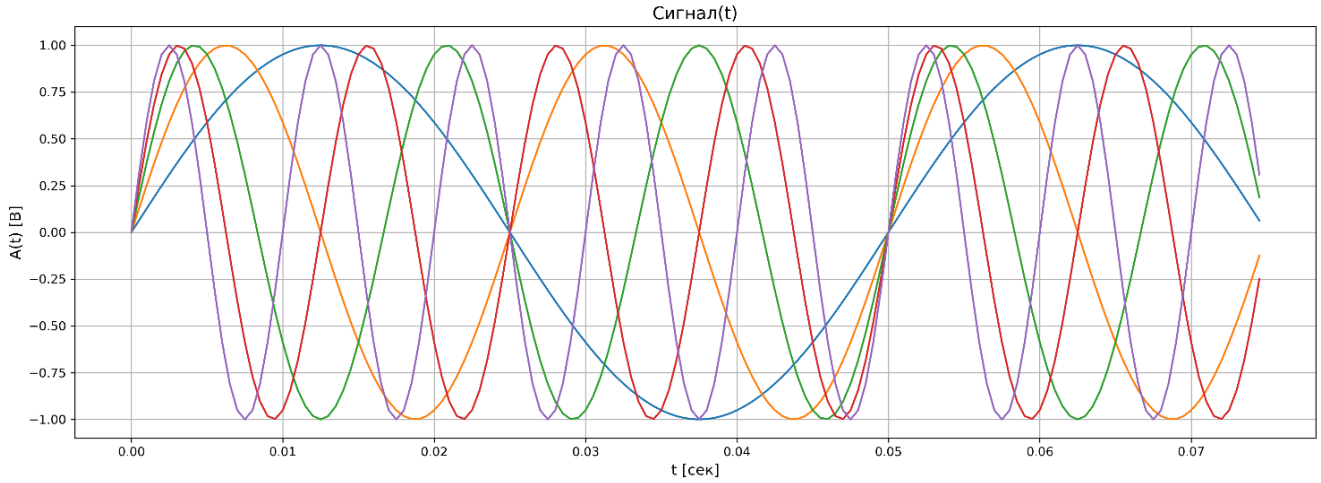

Рисунок 1.1 - Отдельные гармоники сигнала

```
X = 0
for k in np.arange(f.size):
      X += np.sin(2*np.pi*f[k]*T*i)
X *= A/f.size
from matplotlib.ticker import (AutoMinorLocator)

fig, ax = plt.subplots(figsize=(10, 6), dpi=300)
ax.set_title('Полигармонический сигнал X(t)', fontsize=14)
ax.set_xlabel('t [сек]', fontsize=12)
ax.set_ylabel('X(t) [В]', fontsize=12)

ax.plot(i*T, X)
ax.plot([i[0]*T, i[-1]*T], [A, A], 'r-*', linewidth=1, alpha=0.3)
ax.plot([i[0]*T, i[-1]*T], [-A, -A], 'r-*', linewidth=1, alpha=0.3)
ax.xaxis.set_minor_locator(AutoMinorLocator())
ax.yaxis.set_minor_locator(AutoMinorLocator())

ax.grid(which='major', color='blue', linewidth=0.75, alpha=0.3)
ax.grid(which='minor', linestyle='--', color='blue', linewidth=0.5, alpha=0.3)
```

```
fig.savefig(r'fig_1.png')
```

В коде выше:

1) $X \mathrel{+}= np.sin(2 * np.pi * f[k] * T * i)$: на каждой итерации цикла к `X` добавляется синусоида с частотой `f[k]`, дискредитированная с периодом `T` и имеющая отсчеты, заданные массивом `i`.
2) $X \mathrel{*}= A/f.size$: после суммирования всех гармоник, результат умножается на $A/f.size$ для нормировки амплитуды сигнала к заданному значению `A`.
3) $from\ matplotlib.ticker\ import\ (AutoMinorLocator)$: Импортирует класс $AutoMinorLocator$ для автоматического размещения мелких делений на осях.

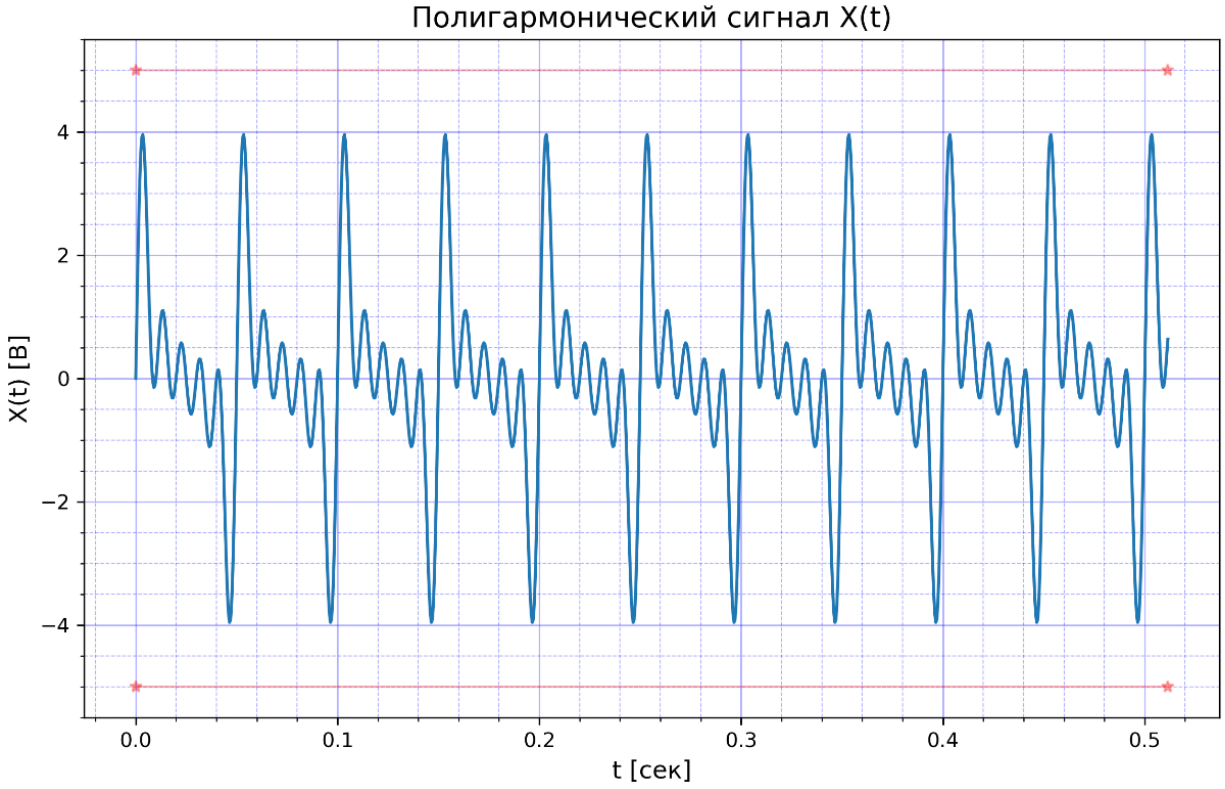

Рисунок 1.2 – Полигармонический сигнал

Полигармонический сигнал крайне сильно зависит от величины своих частот, на рисунке ниже график сигнала с сильно отдаленными друг от друга частотами:

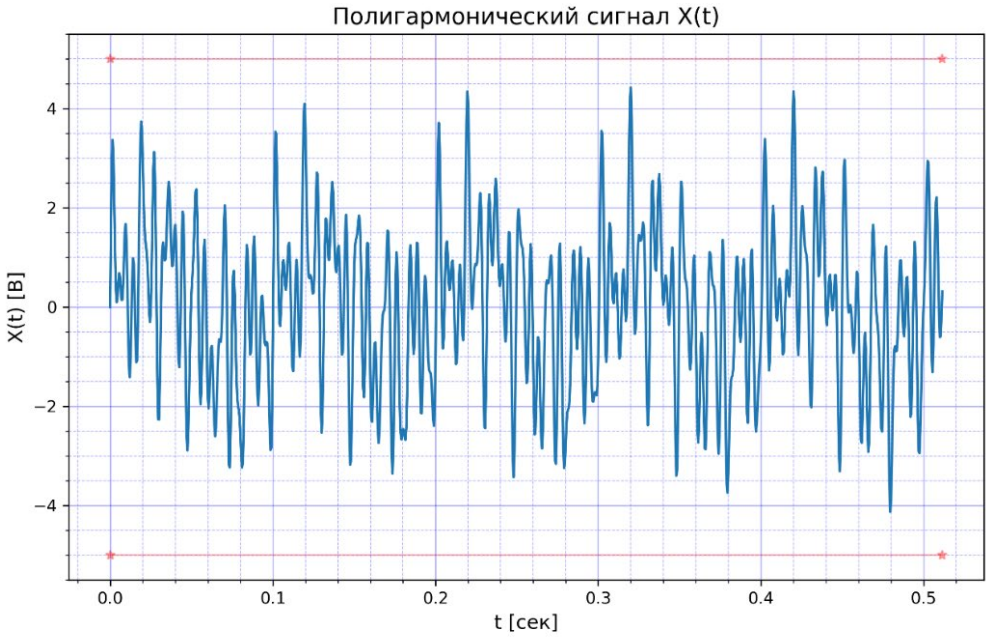

Рисунок 1.3 – Полигармонический сигнал с большим разбросом частот

Теперь рассмотрим, как можно сгенерировать несколько наиболее распространённых видов шумов. Первым рассмотрим нормальный или Гауссовский шум:

```
rng = np.random.default_rng(12340)
noise_continuous = rng.normal(0, A * 0.2, N)
```

Разберем как генерируется нормальный шум:

1) Строка $rng = np.random.default\_rng(12340)$ создает экземпляр генератора случайных чисел rng из библиотеки $NumPy$.
2) $np.random.default\_rng()$ — это рекомендуемый в настоящее время способ создания генераторов случайных чисел в $NumPy$.
3) `12340` — это так называемое "зерно" ($seed$) генератора. Задание фиксированного зерна гарантирует, что при каждом запуске кода с этим зерном будет генерироваться одна и та же последовательность случайных чисел. Это важно для воспроизводимости результатов. Если не задавать зерно, то при каждом запуске будут генерироваться разные случайные числа.

4) $noise\_continuous = rng.normal(0, A * 0.2, N)$: Эта строка генерирует сам непрерывный шум.
5) N: Количество генерируемых случайных чисел. Оно равно количеству отсчетов в исходном сигнале, чтобы шум и сигнал имели одинаковую длину.

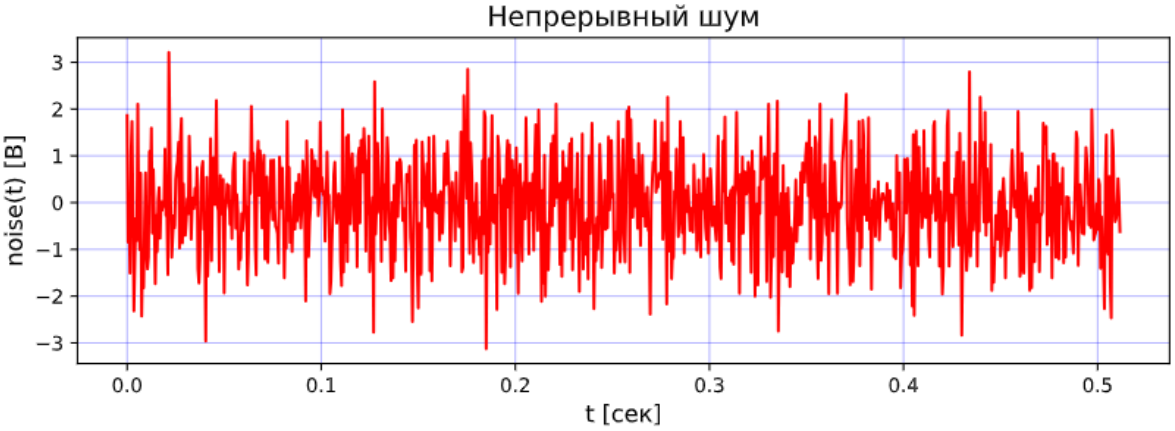

Рисунок 1.4 – Нормальный шум

В результате будет получен белый гауссовский шум со средним значением 0 и стандартным отклонением, равным 20% от амплитуды сигнала.

Теперь сгенерируем прерывистый шум:

```
noise_intermittent = np.zeros(N)
intermittent_start = int(N * 0.3)
intermittent_end = int(N * 0.5)
noise_intermittent[intermittent_start:intermittent_end]=
X[intermittent_start:intermittent_end]
```

Создает массив $noise\_intermittent$ длиной `N`, заполненный нулями, с помощью функции $np.zeros()$. Изначально предполагается, что шума нет. $intermittent\_start = int(N * 0.3)$:

Вычисляет индекс конца "прерывания" сигнала. Он равен 50% от общей длины сигнала `N`. Это ключевая строка, которая моделирует прерывание. С помощью среза выбирается часть массива $noise\_intermittent$, соответствующая интервалу прерывания.

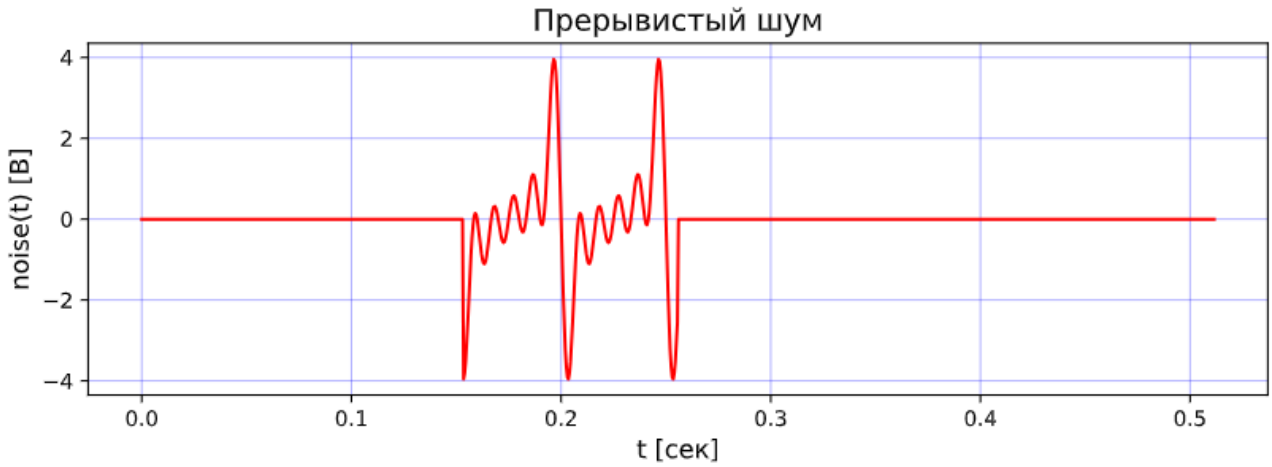

Рисунок 1.5 – Прерывистый шум

В итоге получим сигнал, у которого на интервале от 30% до 50% длины сигнала значения равны инвертированному исходному сигналу, а в остальной части - нули. Это имитирует ситуацию, когда сигнал на некоторое время пропадает.

И последним рассмотрим импульсный шум:

```
noise_impulse = np.zeros(N)
impulse_count = 5

impulse_indices = rng.choice(N, impulse_count, replace=False)   impulse_magnitude = A *
noise_impulse[impulse_indices] = impulse_magnitude * rng.choice([-1, 1], impulse_count)
```

Проанализируем код выше:

1) $impulse\_indices = rng.choice(N, impulse\_count, replace = False)$: Генерирует массив $impulse\_indices$, содержащий $impulse\_count$ (5) случайных уникальных (неповторяющихся) индексов в диапазоне от 0 до `N-1`.
2) $rng.choice(N, impulse\_count, replace = False)$: `N`: задаёт диапазон, из которого выбираются индексы (от 0 до `N-1`).
3) $impulse\_count$: Определяет количество выбираемых индексов.

4) $rng.choice([-1, 1], impulse\_count)$: Генерирует массив из $impulse\_count$ случайных чисел, каждое из которых равно либо -1, либо 1. Это определяет знак (положительный или отрицательный) каждого импульса.
5) Полученный массив случайных знаков умножается на $impulse\_magnitude$, чтобы получить итоговые значения импульсов.

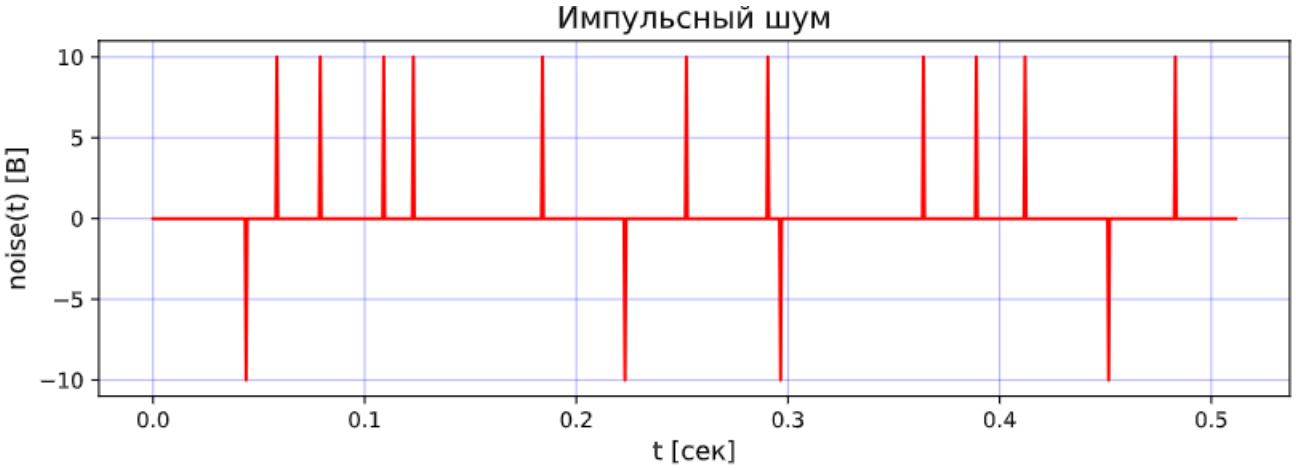

Рисунок 1.6 – Импульсный шум

В результате получим сигнал, в котором в $impulse\_count$ случайных позициях находятся импульсы с величиной, равной удвоенной амплитуде сигнала, и случайным знаком (положительным или отрицательным), а остальные элементы равны нулю.

Для отрисовки шума использовался следующий код:

```
def plot_noise(noise, noise_type, i, T):
      fig, ax = plt.subplots(figsize=(10, 3), dpi=300)
      ax.set_title(f'{noise_type} шум', fontsize=14)
      ax.set_xlabel('t [сек]', fontsize=12)
      ax.set_ylabel('noise(t) [B]', fontsize=12)
      ax.plot(i * T, noise, color='red')
      ax.grid(which='major', color='blue', linewidth=0.75, alpha=0.3)
      ax.grid(which='minor', linestyle='--', color='blue', linewidth=0.5, alpha=0.3)
      plt.show()
```

Рассмотрим данный код подробней:

Теперь наложим нормальный сигнал шум, и визуализируем результат:

```
Xn = X + noise
fig, ax = plt.subplots(figsize=(10, 6), dpi=300)
ax.set_title('Полигармонический сигнал X(t)', fontsize=14)
ax.set_xlabel('t [сек]', fontsize=12)
ax.set_ylabel('X(t), Xn(t) [B]', fontsize=12)
ax.plot(i*T, Xn, label='Сигнал с шумом Xn(t)')
ax.plot(i*T, X, label='Сигнал без шума X(t)')
ax.plot([i[0]*T, i[-1]*T], [A, A], 'r-*', linewidth=1, alpha=0.3)
ax.plot([i[0]*T, i[-1]*T], [-A, -A], 'r-*', linewidth=1, alpha=0.3)
ax.xaxis.set_minor_locator(AutoMinorLocator())
ax.yaxis.set_minor_locator(AutoMinorLocator())
ax.grid(which='major', color='blue', linewidth=0.75, alpha=0.3)
ax.grid(which='minor', linestyle='--', color='blue', linewidth=0.5, alpha=0.3)
ax.legend()
fig.savefig(r'fig_3.png')
```

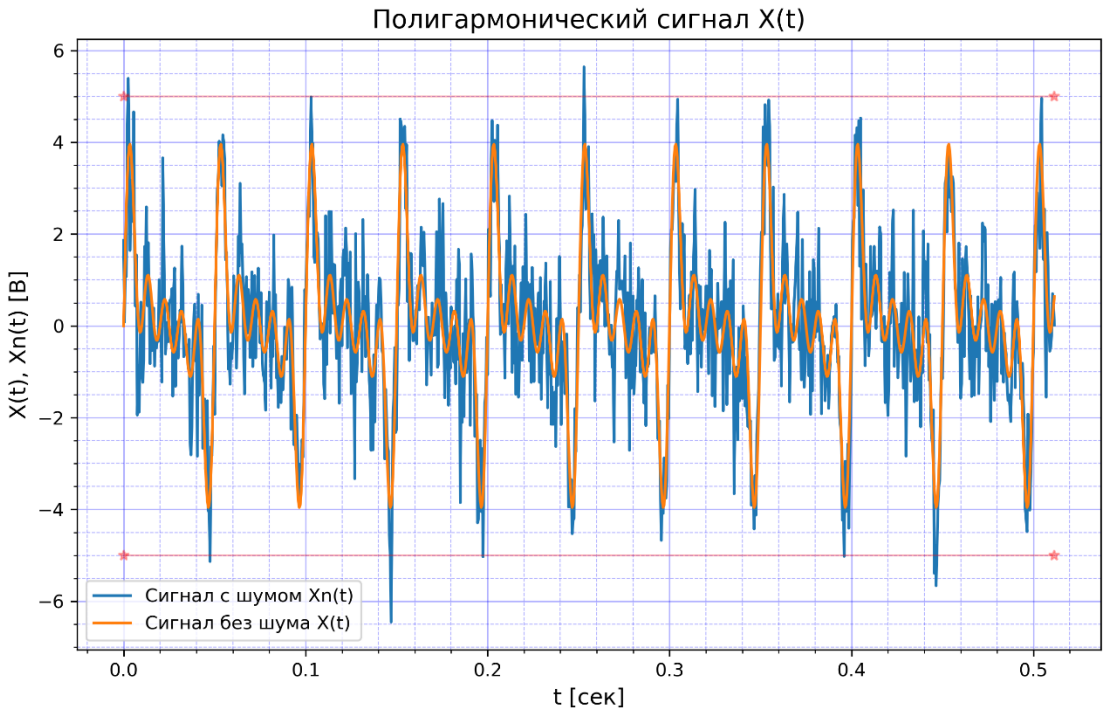

Рисунок 1.7 – Полигармонический сигнал с наложенным шумом

2 Спектральный анализ полигармонического сигнала

2.1 Теоретические сведения

Спектральный анализ полигармонического сигнала — это процесс разложения полигармонического сигнала на его составляющие частоты.

Спектральный анализ полигармонического сигнала позволяет проанализировать его спектральные характеристики, такие как гармоническая структура, наличие резонансных пиков, шумы и другие особенности.

Для анализа выбранного полигармонического сигнала, воспользуемся преобразованием Фурье, определённым на конечном интервале времени. Это преобразование вычисляется, как правило, при помощи алгоритма быстрого преобразования Фурье. Определение преобразования для комплексной функции $z(i), i = 0,1, \dots, N - 1$ имеет в этом случае вид:

$$a_m = \frac{1}{n}\sum_{k=0}^{n-1} A_k \mathrm{e}^{-2\pi i + \frac{mk}{n}}, m = 0, \dots, n-1 \qquad 2.1$$

$$A_k = \frac{1}{n}\sum_{k=0}^{n-1} A_m \mathrm{e}^{-2\pi i + \frac{mk}{n}}, m = 0, \dots, n-1 \qquad 2.2$$

Здесь A_k – комплексная функция, а выражение a_m вычисляет значение этой функции в окрестности данной точки.

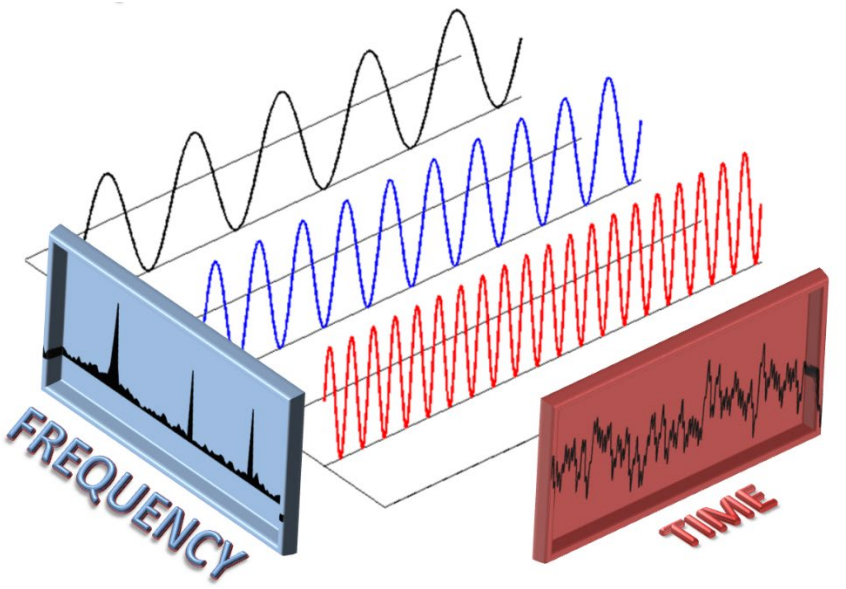

Рисунок 2.1 – Пример наглядного преобразования Фурье

Преобразование Фурье позволяет перевести сигнал из временной области в частотную область, где каждая составляющая частоты представлена комплексным числом, состоящим из действительной и мнимой частей. Амплитуда комплексного числа соответствует величине компоненты частоты, а фаза определяет относительный сдвиг фаз сигнала. Преобразование Фурье описывается следующим образом:

$$X_c(i) = X_\eta(i).* Z(i) \tag{2.3}$$

Процесс выполнения преобразования Фурье включает следующие шаги:

1. Задание входной функции или сигнала, для которого требуется выполнить преобразование Фурье.

2. Вычисление преобразования Фурье с помощью интеграла Фурье или дискретного преобразования Фурье (ДПФ).

3. Построение спектра, который показывает амплитуду и фазу каждой частотной компоненты входного сигнала.

4. Анализ спектра и интерпретация результатов преобразования Фурье с учетом задачи, которую необходимо решить.

Стоит отметить, что для реализации преобразования Фурье, такого как: алгоритм быстрого преобразования Фурье (БПФ) для дискретного случая существует не только библиотека $NumPy$. $Python$, в целом, легко можно связать с $Matlab$ для реализации алгоритмов Фильтрации.

Обратное преобразование Фурье - выполняет обратную операцию прямого преобразования Фурье, преобразуя спектр сигнала из частотной области обратно во временную область. Результатом является восстановленный временной сигнал,

который является обратной к исходному сигналу. Обратное преобразование Фурье также имеет широкий спектр применений, включая синтез звука, восстановление сигналов, фильтрацию и др. Обратное преобразование Фурье описывается следующим образом:

$$x_c(t) = ifft\big(X_c(i)\big) \tag{2.4}$$

Амплитудный спектр представляет собой модуль комплексного спектра, который получаете в результате преобразования Фурье. Он показывает зависимость амплитуды сигнала от частоты, какие частоты присутствуют в сигнале и с какой амплитудой. Благодаря амплитудному спектру, в сигнале можно найти скрытый особенности сигнала, которые не видны во временной области. Например, аддитивный шум.

Центрирование в контексте обработки данных означает приведение среднего значения сигнала к нулю или к определённому центральному значению.

Это полезная операция при обработке сигналов, таких как звуковые или временные ряды, когда нужно сосредоточиться на изменении амплитуды и формы сигнала, а не на его абсолютном смещении. Выполняется для более удобной визуализации и интерпретации спектральных данных.

2.2 Практическая реализация

Осуществим преобразование Фурье с помощью функции $fft.fft$ из библиотеки $NumPy$, для проведения спектрального анализа. Чтобы построить амплитудный спектр, необходимо найти модель комплексного числа, в данном случае модель всего массива.

```
Xf=np.fft.fft(Xn)/len(Xn)
Xf=np.fft.fftn(Xn)
Af=np.abs(Xf)
b=1/(N*T)
Ff=np.arange(N)*b
```

В коде выше интерес представляет $np.fft.fft(Xn)$:

1) $np.fft.fft(Xn)$: это основная функция из библиотеки $NumPy$ для вычисления дискретного преобразования Фурье (ДПФ) одномерного сигнала Xn.
    (a) Вход: Xn – это массив (вектор), представляющий наш сигнал во временной области. Это может быть, например, последовательность

отсчетов звукового сигнала, сигнала с датчика, или любой другой дискретный сигнал.

(b) Выход: $np.fft.fft(Xn)$ возвращает массив комплексных чисел той же длины, что и Xn. Каждое комплексное число в этом массиве представляет амплитуду и фазу определенной частоты в сигнале

(c) $/len(Xn)$: операция нормировки. Зачем нормировать? Без нормировки, амплитуды в спектре Xf будут зависеть от длины сигнала Xn. Чтобы сравнить спектры сигналов разной длины или чтобы корректно проводить обратное преобразование Фурье, обычно спектр нормализуют делением на количество точек `N` (т.е. $len(Xn)$). Это делает амплитуды спектра сопоставимыми с амплитудами исходного сигнала.

2) $Xf = np.fft.fftn(Xn)$: Эта функция вычисляет многомерное ДПФ. Она используется, когда ваш сигнал представляет собой многомерный массив, например, изображение (двумерный массив), трехмерный объем данных, и т.д.

(a) Вход: Xn – это многомерный массив.

(b) Выход: $np.fft.fftn(Xn)$ возвращает многомерный массив комплексных чисел, представляющий спектр входного массива.

3) $Af = np.abs(Xf)$: Это функция для вычисления модуля (абсолютного значения) комплексных чисел.

(a) Вход: Xf – массив комплексных чисел, который является результатом прямого преобразования Фурье.

(b) Выход: Af – массив действительных чисел, представляющий амплитудный спектр сигнала. Каждое число в Af – это амплитуда соответствующей частотной компоненты в сигнале.

4) $b = 1/(N * T)$

i) b – это разрешение по частоте (ширина одного частотного "бина" или дискретного значения частоты) в частотном спектре.

(a) `N`: Количество отсчетов (точек) в исходном сигнале Xn.

(b) `T`: Общая длительность сигнала во временной области (в секундах, или других единицах времени).

(c) Единицы измерения: b измеряется в герцах (Гц) или в тех же единицах частоты, что и сигнал во временной области. Дискретное преобразование Фурье работает с дискретным сигналом и выдает набор дискретных частот. Соответственно, разрешение по частоте показывает нам, какое расстояние в частотной области будет между соседними частотами, на которые разложен наш сигнал. Чем меньше b, тем более точно можно проанализировать сигнал на предмет наличия частот в его составе.

5) $Ff = np.arange(N) * b$

i) $np.arange(N)$: Создает массив целых чисел от 0 до `N-1`

6) b: умножает каждое целое число на разрешение по частоте `b`.

(a) Ff — это массив, содержащий частоты, соответствующие каждому бину (точке) в амплитудном спектре Af. То есть Ff является массивом частот, на которые разложен ваш сигнал.

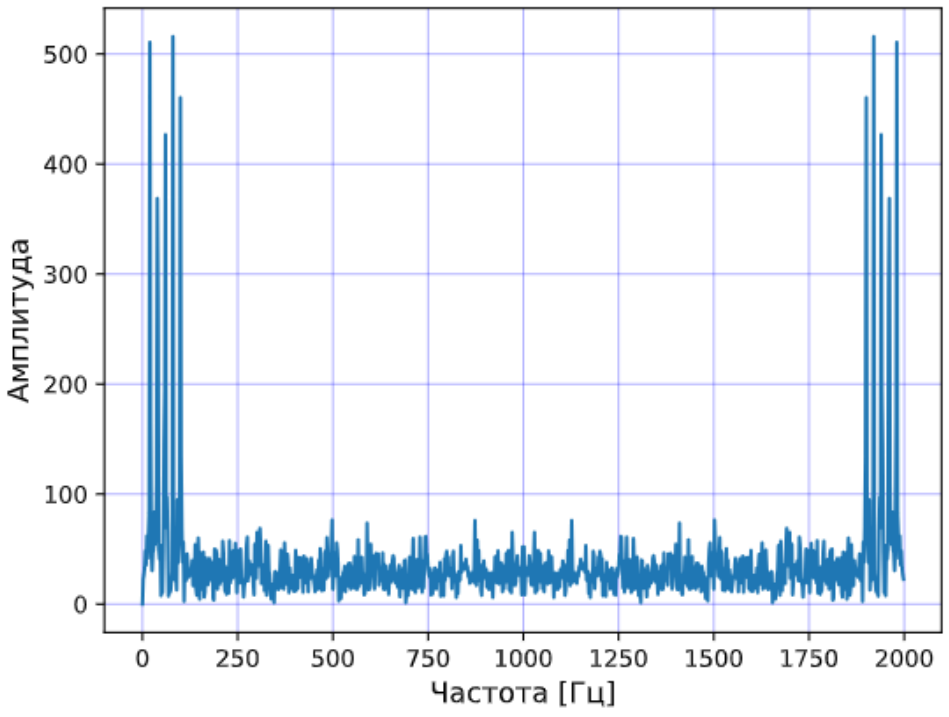

Рисунок 2.2 – Сигнал в спектральной области

В итоге мы перешли из временной в частотную область, вычислили разрешение по частоте, которое показывает, насколько точно разложен спектр сигнала. В результате ДПФ (без перестановки) первая половина спектра $(0 \dots N/2)$ представляет положительные частоты, а вторая половина $(N/2 \dots N-1)$ – отрицательные (которые в большинстве случаев не используются).

На рисунке 2.2 уже можно качественно оценить, сколько амплитуд присутствуют в зашумленном сигнале, острые пики, сильно больше среднего значения массива данных этого графика. Приблизим график, чтобы увидеть, как именно изменятся значения сигнала на этих амплитудах:

```
plt.plot(Ff[0:100], Af[0:100])
plt.grid(which='major', color='blue', linewidth=0.75, alpha=0.3)
plt.xlabel("Частота [Гц]", fontsize=12)
plt.ylabel("Амплитуда", fontsize=12)
plt.grid(True)
plt.show()
```

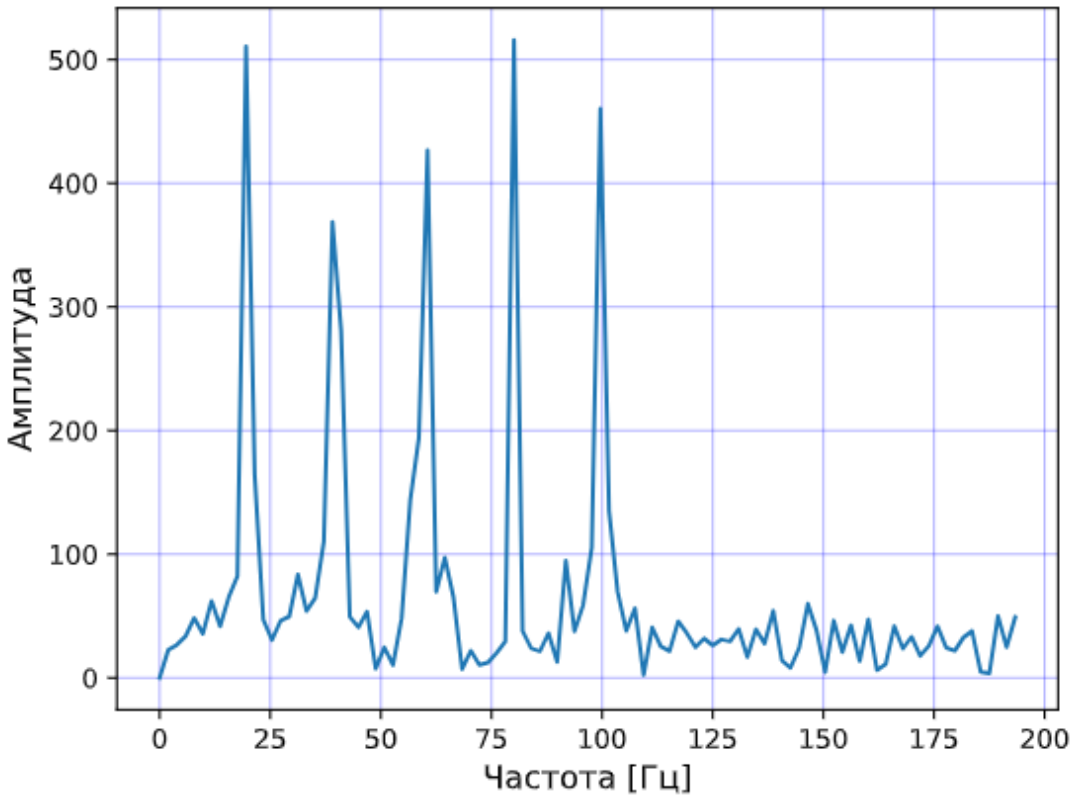

Рисунок 2.3 – Приближенные амплитуды рассматриваемого сигнала

3 Реализация традиционных алгоритмов фильтрации

3.1 Фильтр низких частот

3.1.1 Теоретические сведения

Фильтр сигнала – это устройство или алгоритм, применяемый для изменения спектра (частотного состава) сигнала путем удаления нежелательных компонентов или усиления нужных компонентов. Он применяется в обработке сигналов для различных целей, таких как устранение шумов, снижение искажений, выделение конкретных частот и т. д.

Уровень отсечения фильтрации — это частота или амплитуда, при достижении или превышении которой фильтр начинает уменьшать амплитуду сигнала или его частотные компоненты. Это значит, что сигналы или компоненты, превышающие уровень отсечения, будут подавлены или отфильтрованы, в то время как те, что находятся ниже уровня отсечения, могут быть сохранены или пропущены без изменений.

$$Z(i) = \begin{cases} 0, A_X(i) < L \\ 1, A_X(i) \geq L \end{cases} \quad (3.1)$$

Метод фильтрации полигармонического сигнала заключается в выборе уровня фильтрации для выделения полезного сигнала. После выбора уровня фильтрации, необходимо будет выделить из амплитудного спектра гармоники, которые находятся выше уровня фильтра. Значения ниже уровня фильтра будут отброшены из сигнала. Тем самым из зашумлённого сигнал выделится полезный сигнал.

3.1.2 Практическая реализация

Фильтрацию условно можно разделить на несколько шагов:

1) Перевод сигнала в спектральную область. Данное преобразование подробно рассматривалось в разделе выше.
2) Определение порога фильтрации:

```
L = np.max(Af[Af < np.max(Af)/2]) * 0.95
```

Находим максимальное значение среди амплитуд, которые меньше половины максимальной амплитуды. Умножаем на 0.95 для создания небольшого запаса.

3) Визуализация спектра и порога:

```
fig, ax = plt.subplots()
ax.plot(Ff, Af)  # График спектра
ax.plot([Ff[0], Ff[-1]], [L, L], 'r-', alpha=0.5)  # Горизонтальная линия порога
```

Отображаем амплитудный спектр сигнала. Показываем уровень порога красной линией

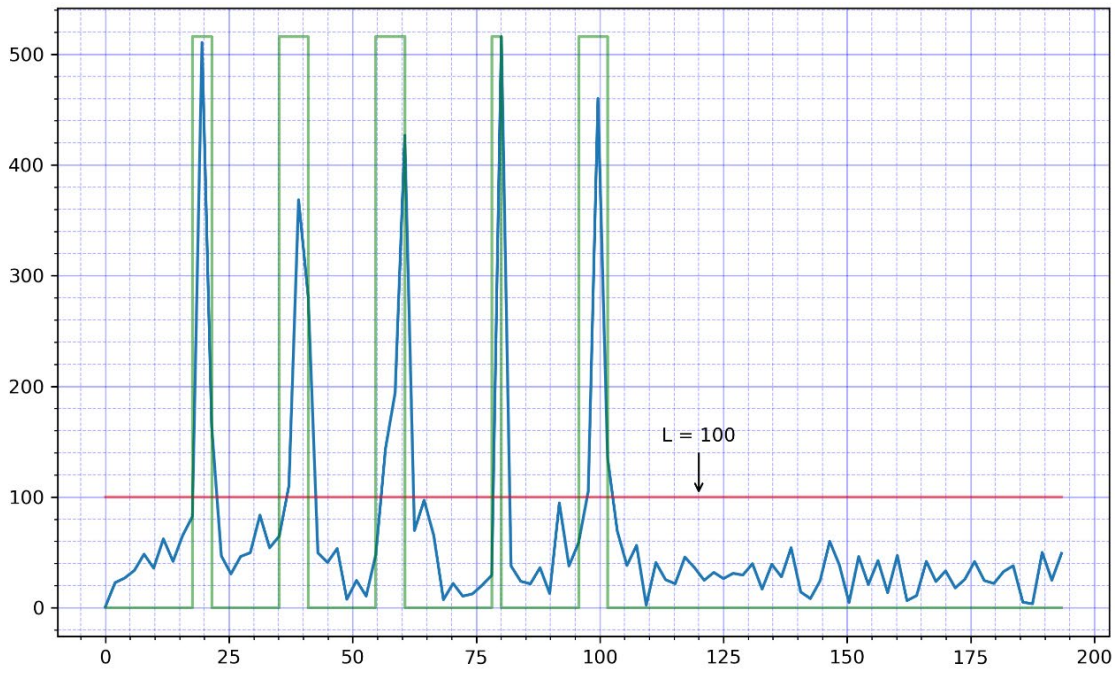

Рисунок 3.1 – Амплитудный спектр с порогом

4) Создаем частотное окна:

```
Wf = np.where(Af>L, 1, 0)
```

Создаем бинарную маску: 1 для частот с амплитудой выше порога, 0 для остальных значений.

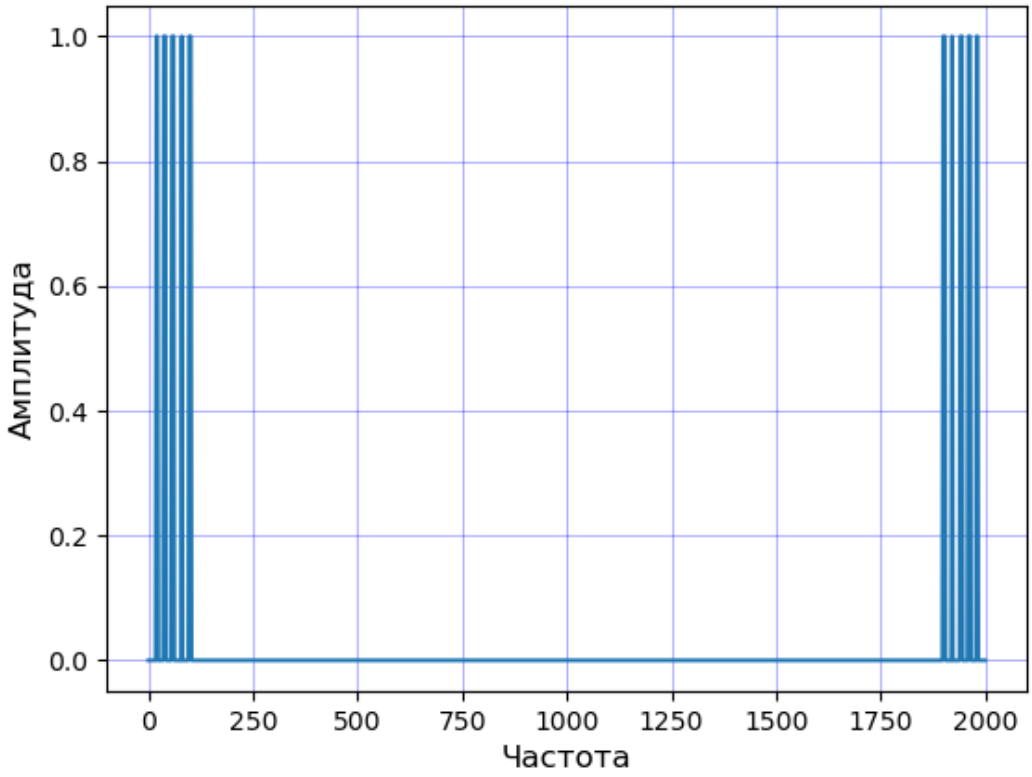

Рисунок 3.2 – Маска фильтра - выделение амплитуд на всей длине

5) Применение фильтра:

```
Xf0 = Xf * Wf
```

Умножает спектр сигнала на частотную маску. Обнуляет все частотные компоненты ниже порога

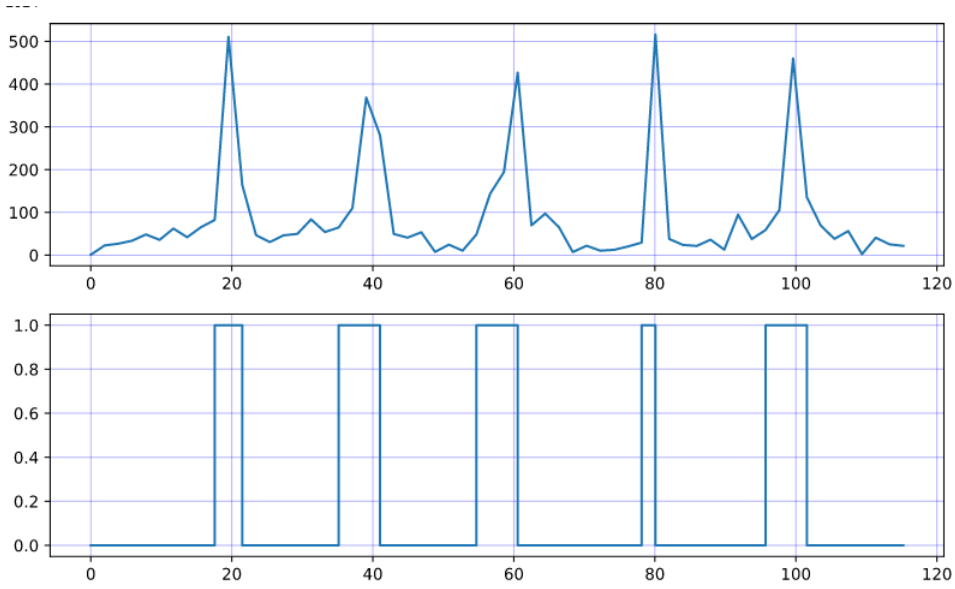

Рисунок 3.3 – Графики АЧХ и фильтра

6) Выполняем преобразование:

```
c0 = np.real(np.fft.ifft(Xf0))/T
c0 = np.real(np.fft.ifftn(Xf0))
```

Выполняет обратное преобразование Фурье. Берет только действительную часть результата. Нормализует результат делением на период сигнала.

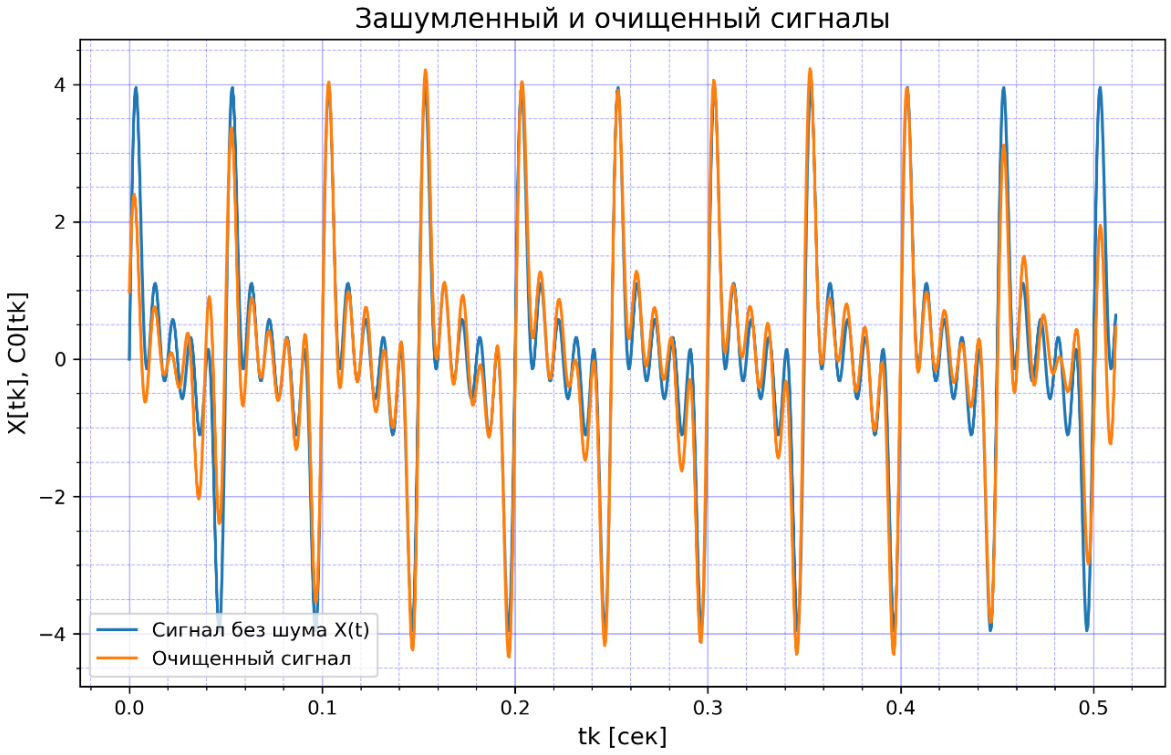

Рисунок 3.4 – Исходный и очищенный сигналы

Этот фильтр работает как простой пороговый фильтр в частотной области, удаляя все частотные компоненты, амплитуда которых ниже определенного порога. Это эффективно для удаления высокочастотного шума, так как высокочастотные компоненты обычно имеют меньшую амплитуду. Его главная проблема — это плохой показатель фильтрации при импульсном шуме, а также большая вычислительная нагрузка из-за преобразования в спектральную область.

3.2 Алгоритм скользящего среднего

Простое скользящее среднее (SMA) — это невзвешенное среднее предыдущих k значений. В некоторых реализациях среднее значение берется из одинакового числа точек по обе стороны от рассматриваемого значения. Благодаря этому получается избежать смещения полученных значений по времени относительно исходных данных. Пусть $p_1, p_2, \dots p_n$. Среднее значение k точки при числе имеющихся значений n можно рассчитать, как:

$$SMA_k = \frac{p_{n-k+1} + p_{n-k+2} + \cdots + p_n}{k} = = \frac{1}{k} \sum_{i=n-k+2}^{n+1} p_i \tag{3.2}$$

Большим плюсом данного алгоритма является то, что при расчете следующего значения на том же наборе значений, новое значение p_{n+1} входит в выборку, а p_{n-k+1} выбывает, что упрощает вычисления сводя все операции, требуемые на одну итерацию к 3 (два сложения, одно деление):

$$SMA_{k+1} = \frac{1}{k} \sum_{i=n-k+2}^{n+1} p_i =$$

$$= \frac{1}{k}(p_{n-k+2} + p_{n-k+3} + \cdots + p_n + p_{n+1} + p_{n-k+1} + p_{n-k+1}) =$$

$$= SMA_k + \frac{1}{k}(p_{n+1} - p_{n-k+1}) \tag{3.3}$$

Таблица 3.1 – Зависимость результата фильтрации от размера окна фильтрации

Размер окна фильтрации	R2 ошибка для высоко частотного сигнала	R2 ошибка для низкочастотного сигнала
5	0.9974	0.9817
10	0.9955	0.9909
15	0.9909	0.9940

Как видно из таблицы 3.1 на данный фильтр сильно влияет вид сигнала, от которого зависит оптимальное значение параметра размера окна фильтрации.

Если используемые данные не сгруппированы по среднему значению, то простая скользящая средняя запаздывает по отношению к последним данным на половину периода выборки. На SMA также может оказывать непропорционально большое влияние выпадение старых данных или поступление новых. Одной из характеристик SMA является то, что если данные имеют периодические колебания, то применение SMA того же периода устранит эти колебания (среднее значение всегда содержит один полный цикл).

Основным недостатком скользящего среднего является то, что оно пропускает значительное количество сигналов, длительность которых меньше, чем длина окна. Хуже того, оно их инвертирует. Это может привести к неожиданным артефактам, таким как появление пиков в сглаженном результате там, где в исходных данных были впадины. Это также приводит к тому, что результат получается менее гладким, чем ожидалось, поскольку некоторые высокие частоты не удаляются должным образом.

Теперь рассмотрим практическую реализацию алгоритма фильтрации:

Инициализация буфера:

При первом вызове функции $arith\_mean$, проверяется, существует ли атрибут $buffer$ у функции. Если не существует, создается буфер размером $buffer\_size$, заполненный значением `f`. Это позволяет функции "запомнить" предыдущие значения для вычисления среднего.

```
if not hasattr(arith_mean, "buffer"):
    arith_mean.buffer = [f] * buffer_size
```

Буфер сдвигается влево, удаляя самое старое значение, и добавляется новое значение `f` в конец буфера. Таким образом, буфер всегда содержит последние $buffer\_size$ значений.

```
mean = 0
for e in arith_mean.buffer: mean += e
    mean /= len(arith_mean.buffer)
```

Суммируются все элементы буфера, а затем сумма делится на количество элементов в буфере для получения среднего значения.

Экспоненциальная скользящая средняя

Этот фильтр является модификацией рассмотренного выше алгоритма. Его ключевая разница заключается в том, что важность каждого последующего вычисляемого значения увеличивается, что делает его крайне полезным при сигналах, имеющих возрастающую природу:

$$EMA_{next,k} = \mathrm{p_k} \times \mathrm{k} + \mathrm{EMA_k} \times (1 - \mathrm{k}) \quad (3.4)$$

Для проверки его эффективности выведем график абсолютной ошибки между исходным сигналом и очищенным в результате фильтрации с помощью SMA и EMA (код фильтрации приложение А):

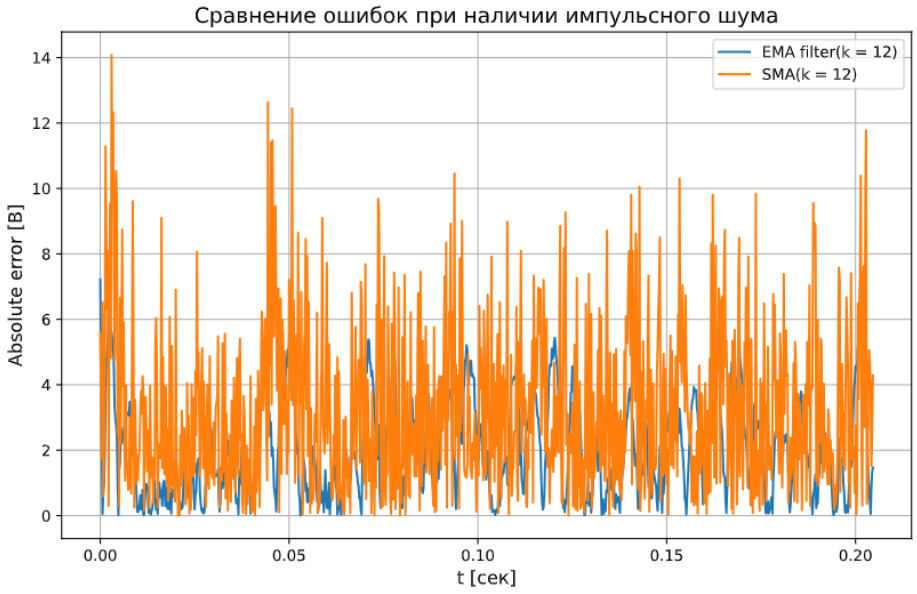

Рисунок 3.5 – Графики абсолютных ошибок между очищенными значениями и исходными

Как видно, алгоритм EMA дает несколько лучшие результаты чем SME, но из-за его свойств отдавать большее значение последующим значениям он подходит далеко не для всех сигналов.

3.3 Фильтр Винера

3.3.1 Теоретические сведения

Пусть мы имеет набор данных из значений $x_1, x_2, \dots, x_N$, составляющие шума, подчиняющиеся нормальному распределению, обозначим как $n_1, n_2, \dots, n_N$, тогда зашумленный сигнал представляет из себя их сумму $z_k = x_k + n_k, k = 1,2, \dots, N$. Задача получить оценки $\overline{x}_k$ максимально близкие к истинным данным x_k. Критерием качества для подобной задачи удобно выбрать:

$$E = M\left\{\left(x_k - \overline{x}_k\right)^2\right\} \to \min \tag{3.5}$$

Оценки же находятся следующим образом:

$$\overline{x}_k = a_{k1}z_1 + a_{k2}z_2 + \cdots + a_{kN}z_N = \sum_{i=1}^{N} a_{ki}z_i = \overline{a_k}^T \bar{z} \tag{3.6}$$

Видно, что для вычисления оценки необходимо получить значения весовых коэффициентов a_{ki}, для их нахождения продифференцируем выражение E по $\overline{a_k}^T$:

$$\frac{dE(\overline{a_k})}{d\overline{a_k}^T} = \frac{dM\left\{x_k - \overline{a_k}^T \bar{z}\right\}}{d\overline{a_k}^T} = -2M\left\{\left(x_k - \overline{a_k}^T \bar{z}\right)\bar{z}^T\right\} = 0 \tag{3.7}$$

$$M\left\{x_k \bar{z}^T\right\} - \overline{a_k}^T M\{(\bar{z}\bar{z}^T)\} = 0 \tag{3.8}$$

Распишем математические ожидания:

$$M\{x_k \bar{z}^T\} = M\left\{x_k \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_k \\ \dots \\ x_N \end{bmatrix}\right\} + M\left\{x_k \begin{bmatrix} n_1 \\ n_2 \\ \dots \\ n_k \\ \dots \\ n_N \end{bmatrix}\right\} = \sigma_x^2 \begin{bmatrix} r_{1k} \\ r_{2k} \\ \dots \\ 1 \\ \dots \\ r_{Nk} \end{bmatrix} = P_k \tag{3.9}$$

Первое математическое ожидание в формуле (3.9) по своей сути корреляция между x_k и остальными данными, обозначим ее, как r_{ik}. Второе математическое ожидание в формуле (3.9) равно 0, так как мы считаем, что корреляции между исходным сигналом и шумами нет.

$$M\{\bar{z}\bar{z}^T\} = M\left\{\begin{bmatrix} z_1 \\ z_2 \\ \cdots \\ z_N \end{bmatrix}[z_1 \quad z_2 \quad \cdots \quad z_N]\right\} =$$

$$= \left\{\begin{bmatrix} x_1 \\ x_2 \\ \cdots \\ x_N \end{bmatrix}[x_1 \quad x_2 \quad \cdots \quad x_N]\right\} + \left\{\begin{bmatrix} n_1 \\ n_2 \\ \cdots \\ n_N \end{bmatrix}[n_1 \quad n_2 \quad \cdots \quad n_N]\right\} \tag{3.10}$$

В формуле (3.9) математические ожидания образуют две корреляционные матрицы:

$$R = \sigma_x^2 \begin{bmatrix} 1 & r_{12} & \cdots & r_{1N} \\ r_{21} & 1 & \cdots & r_{2N} \\ \cdot & \cdot & \cdot & \cdot \\ r_{N1} & r_{N2} & \ldots & 1 \end{bmatrix}; V = \begin{bmatrix} \sigma_x^2 & 0 & 0 & 0 \\ 0 & \sigma_x^2 & 0 & 0 \\ \cdot & \cdot & \cdot & \cdot \\ 0 & 0 & 0 & \sigma_x^2 \end{bmatrix} \tag{3.11}$$

Теперь выражение (2.2) можно записать следующим образом:

$$\sigma_x^2 P_k - \overline{a_k}^T(\sigma_x^2 R + V) = 0$$

$$P_k - \overline{a_k}^T\left(R + \frac{V}{\sigma_x^2}\sigma_x^2\right) = 0$$

Откуда

$$\overline{a_k}^T = P_k^T\left(R + \frac{V}{\sigma_x^2}\right)^{-1} \tag{3.12}$$

3.3.2 Практическая реализация

Теперь реализуем данный алгоритм и сравним его эффективность с сравнение с рассмотренным ранее EMA:

Инициализируем выходной массив:

```
filtered_data = np.zeros_like(data)
half_window = window_size // 2
```

Создается массив *filtered_data* той же длины, что и входной массив *data*, заполненный нулями. Переменная *half_window* хранит половину размера окна фильтра (округленную вниз), что будет использоваться для определения границ окна.

```
for i in range(len(data)):
    start = max(0, i - half_window)
    end = min(len(data), i + half_window + 1)
```

Для каждого элемента $data[i]$ определяются границы окна фильтра, в котором будет проводиться локальная оценка.

Извлечем текущее окно и оценим локальную мощность сигнала и шума:

```
window = data[start:end]
local_mean = np.mean(window)
local_variance = np.var(window)
noise_variance = np.mean(np.square(window - local_mean))
```

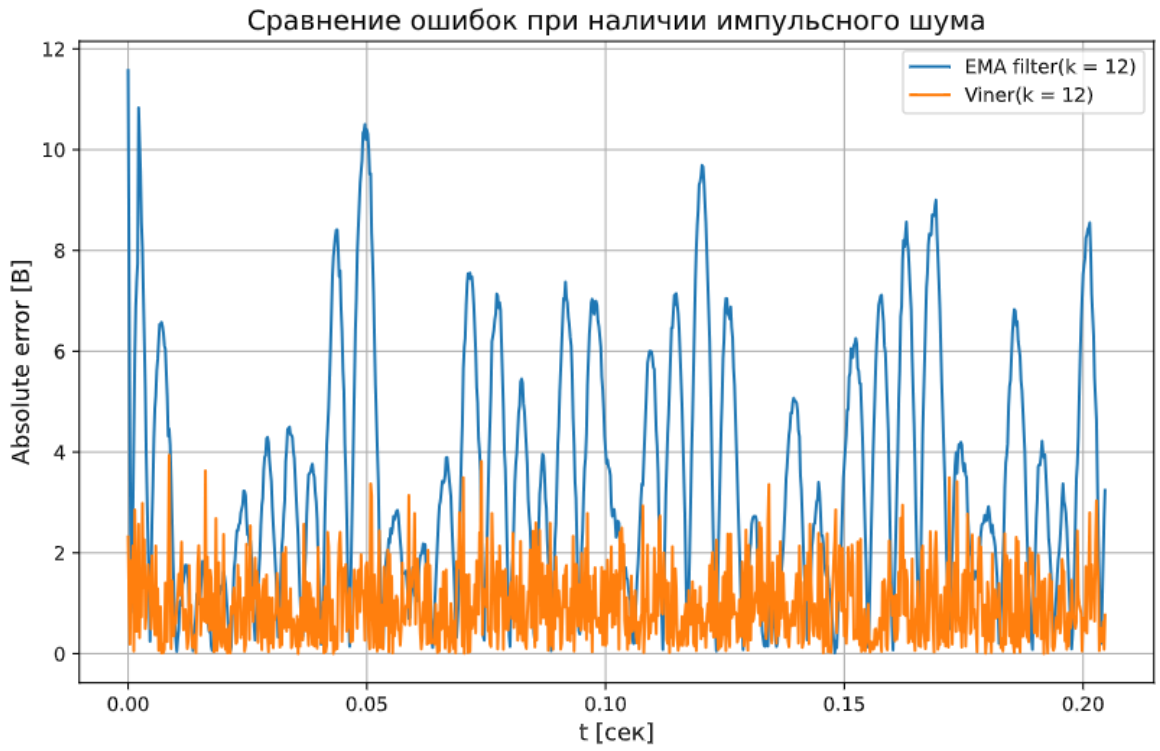

Рисунок 3.6 – Графики абсолютных ошибок между очищенными значениями и исходными

В результате видно, что фильтр справляется лучше, чем скользящая средняя. Его недостатки — это большая вычислительная нагрузка, а также необходимость иметь сразу весь зашумленный сигнал сразу, когда для предыдущего алгоритма достаточно было данных анализируемого окна.

4 Автоэнкодер

4.1 Теоретические сведения

Автоэнкодеры — это архитектура нейронной сети, состоящая из двух частей: *кодировщика,* который кодирует некоторые входные данные в закодированное состояние, и *декодера,* который может декодировать закодированное состояние в другой формат.

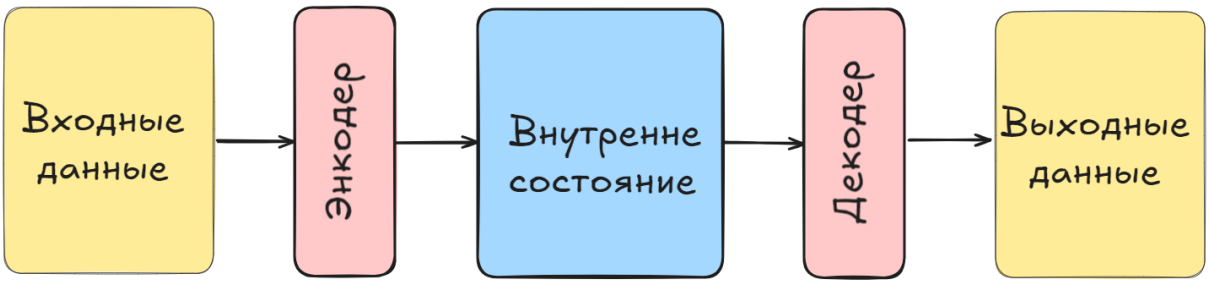

Рисунок 4.1– Структурная схема автоэнкодера

Автоэнкодеры изучают *закодированное состояние* с помощью *кодера* и учатся декодировать это состояние во *что-то еще* с помощью *декодера.* В контексте сигнального шума: предположим, что вы передаете нейронной сети зашумленные данные в качестве *функций,* в то время как чистые данные доступны в качестве *целей*. Следуя рисунку выше, нейронная сеть изучит закодированное состояние на основе зашумленного изображения и попытается декодировать его, чтобы наилучшим образом соответствовать *чистым данным.* Что стоит между чистыми данными и зашумленными данными? Шум. По сути, автоэнкодер научится распознавать шум и удалять его из входного изображения. Большое преимущество такого подхода в том, что теоретически возможно натренировать автоэнкодер обнаруживать шум, не подчиняющийся нормальному распределению, а имеющему сложную природу (прерывистый, импульсный и т. д.).

Это свойство связано с тем, что автоэнкодеры очень естественны для выполнения оценки функций распределения. Объяснить это можно следующим образом: обучающий набор данных определяется плотностью их распределения. Чем выше плотность обучающих примеров вокруг локальной точки во входном

пространстве, тем лучше автоэнкодер реконструирует входной вектор в этом месте пространства. Кроме того, внутри автоэнкодера есть вектор латентного представления входных данных (как правило, низкой размерности), и если данные проецируются в латентном пространстве в область, ранее не задействованную при обучении, то, значит, и в обучающей выборке не было ничего похожего.

Математически это можно обосновать следующим образом.

Автоэнкодер - последовательное применение функции энкодера $z = g(x)$ и декодера $x^* = f(z)$, где x — входной вектор, а z — латентное представление. В некотором подмножестве входных данных (обычно близкое к обучающему), где — невязка. Невязку примем за шум, в грубом приближении как подчиняющийся нормальному распределению (его параметры можно оценить после обучения автоэнкодера). В итоге делается ряд достаточно сильных предположений:

1) невязка — Гаусовский шум
2) автоэнкодер уже «натренирован» и работает

Теперь получим оценку плотности вероятности входного вектора:

$$p(x) = \int_z p(x|z)p(z)dz \tag{4.1}$$

Нам нужно получить связь $p(x)$ и $p(z)$. Для некоторых автоэнкодеров $p(z)$ задается на стадии их тренировки, для других $p(z)$ получить все же проще за счет меньшей размерности Z.

Плотность распределения невязки n известна, значит:

$$p(n) = \text{const} * \exp\left(-\frac{\big(x - f(z)\big)^T\big(x - f(z)\big)}{2\sigma^2}\right) = p(x|z) \tag{4.2}$$

$\big(x - f(z)\big)^T\big(x - f(z)\big)$ – это дистанция между x и его проекцией x^* в некоторой точке z^* это дистанция достигнет своего минимума. В этой точке частные производные аргумента экспоненты в формуле выше будут нулевыми:

$$0 = \frac{\partial f(z^*)^T}{\partial z_i}\big(x - f(z^*)\big) + \big(x - f(z^*)\big)^T\frac{\partial f(z^*)^T}{\partial z_i} \tag{4.3}$$

Здесь $\frac{\partial f(z^*)^T}{\partial z_i}\big(x - f(z^*)\big)$ скаляр, тогда:

$$0 = \frac{\partial f(z^*)^T}{\partial z_i}\big(x - f(z^*)\big) \tag{4.4}$$

Выбор точки z^*, где дистанция $\big(x - f(z)\big)^T\big(x - f(z)\big)$ минимальна, обусловлен процессом оптимизации автоэнкодера. Во время обучения минимизируется квадратичная невязка: $L2_{norm}(x - f_\theta(g_\theta(x))$, где θ – веса автоэнкодера. Т. е. после обучения $g(x) \to z^*$.

Также можно разложить $f(z)$ в ряд Тейлора (до первого члена) вокруг z^*.

$$f(z) = f(z^*) + \nabla f(z^*)(z - z^*) + o\big((z - z^*)\big) \tag{4.5}$$

Теперь уравнение 4 примет вид:

$$p(x|z) \approx const \times \exp\left(-\frac{\Big(\big(x - f(z^*)\big) - \nabla f(z^*)(z - z^*)\Big)^T\Big(\big(x - f(z^*)\big) - \nabla f(z^*)(z - z^*)\Big)}{2\sigma^2}\right) \tag{4.6}$$

Предположим, что $p(z)$ достаточно гладкая функция и не сильно меняется в окрестности $z*$, т.е. заменим $p(z) \to p(z^*)$. Тогда интеграл в формуле (4) примет вид:

$$p(x) = \text{const} * \text{p}(\text{z}^*) * \exp\left(-\frac{\big(x - f(z)\big)^T\big(x - f(z)\big)}{2\sigma^2}\right)\int_z \exp(-(z - z^*)^T W(x)^T W(x)(z - z^*)dz \tag{4.7}$$

где $W(x) = \frac{\nabla f(z^*)}{\sigma}$

Последний интеграл – это n – мерный интеграл Эйлера-Пуассона:

$$\int_z \exp(-(z - z^*)^T W(x)^T W(x)(z - z^*)dz = \sqrt{\frac{1}{\det\left(\frac{W(x)^T W(x)}{2\pi}\right)}} \tag{4.8}$$

Из формулы (4.8) можно сделать следующие выводы:

1) Дистанции между входным вектором и его реконструкцией, чем хуже восстановили — тем меньше $p(x)$

2) Плотности вероятности $p(z^*)$ в точке $z^* = g(x)$
3) Нормировки функции $p(z)$ в точке z^*, которая рассчитывается для автоэнкодера из частных производных функции f.

4.2 Практическая реализация

Основная цель - создание архитектуры автоэнкодера для шумоподавления сигналов. На входе мы будем нормировать и преобразовывать в спектральную область исходный сигнал, а также применим простейший фильтр низких частот, чтобы нейросеть в первую очередь училась распознавать сложные составляющие шумы, которые трудно очистить обычными средствами (полный код архитектуры в приложении Б).

Основной частью модели являются последовательность сверточных и транспонированных сверточных слоев:

```
x = tf.keras.layers.Conv1D(128, kernel_size=4,
kernel_constraint=tf.keras.constraints.MaxNorm(max_norm_value), activation='relu',
kernel_initializer='he_uniform', name='conv1')(reshaped_fft)

x = tf.keras.layers.Conv1D(32, kernel_size=4,
kernel_constraint=tf.keras.constraints.MaxNorm(max_norm_value), activation='relu',
kernel_initializer='he_uniform', name='conv2')(x)
x = tf.keras.layers.Conv1DTranspose(32, kernel_size=4,
kernel_constraint=tf.keras.constraints.MaxNorm(max_norm_value), activation='relu',
kernel_initializer='he_uniform', name='convT1')(x)
x = tf.keras.layers.Conv1D(128, kernel_size=4,
kernel_constraint=tf.keras.constraints.MaxNorm(max_norm_value), activation='relu',
kernel_initializer='he_uniform', name='conv3')(x)
x = tf.keras.layers.Conv1DTranspose(128, kernel_size=4,
kernel_constraint=tf.keras.constraints.MaxNorm(max_norm_value), activation='relu',
kernel_initializer='he_uniform', name="convT2")(x)
x = tf.keras.layers.Conv1DTranspose(128, kernel_size=4,
kernel_constraint=tf.keras.constraints.MaxNorm(max_norm_value), activation='relu',
kernel_initializer='he_uniform', name='convT3')(x)
```

```
x = tf.keras.layers.Conv1D(1, kernel_size=4,
kernel_constraint=tf.keras.constraints.MaxNorm(max_norm_value), activation='sigmoid',
padding='same', name='conv4')(x)
```

Благодаря им реализуется архитектура, представленная на рисунке ниже:

Layer (type)	Output Shape	Param #
input_layer_39 (InputLayer)	(None, 64, 1)	0
fft_layer_71 (FFTLayer)	(None, 64, 1, 1)	0
low_pass_filter_15 (LowPassFilter)	(None, 64, 1, 1)	0
reshape_68 (Reshape)	(None, 64, 1)	0
conv1 (Conv1D)	(None, 61, 128)	640
conv2 (Conv1D)	(None, 58, 32)	16,416
convT1 (Conv1DTranspose)	(None, 61, 32)	4,128
conv3 (Conv1D)	(None, 58, 128)	16,512
convT2 (Conv1DTranspose)	(None, 61, 128)	65,664
convT3 (Conv1DTranspose)	(None, 64, 128)	65,664
conv4 (Conv1D)	(None, 64, 1)	513
fft_layer_72 (FFTLayer)	(None, 64, 1, 1)	0
reshape_69 (Reshape)	(None, 64, 1)	0

Рисунок 4.2 – Архитектура сети

Приведем небольшие пояснения по функциональности слоев в архитектуре сети:

1) $Conv1D$: классическая сверточная операция для одномерных данных. Здесь используется активация $ReLU$ и инициализация весов $He\_uniform$.
2) Первый слой извлекает 128 признаков из входного сигнала.
3) Второй слой уменьшает размерность до 32 признаков.
4) $Conv1DTranspose$: транспонированная свертка, которая увеличивает размерность данных. Используется для восстановления сигнала после обработки. Восстанавливает размерность сигнала до 128 признаков на выходе.
5) Последний слой:

6) Сверточный слой с ядром размером 4.
7) Активация $sigmoid$ для получения значений в диапазоне `[0, 1]`. `padding='same'` сохраняет исходную размерность выхода.
8) Слои ftt_layer, $reshape$, low_pass_filter – являются пользовательскими и отвечают за функционал прямого и обратного преобразования Фурье, сохранение верной размерности внутри модели и фильтрацию низко частного шума, соответственно.

Два слоя Conv1D служат кодировщиком *и* изучают 128 и 32 фильтра соответственно. Они активируются с помощью функции активации $ReLU$ и, как следствие, требуют инициализации He . К каждому из них применяется макс-нормальная регуляризация.

Два слоя Conv1DTranspose, которые изучают фильтры 32 и 128, служат декодером. Они также используют активацию $ReLU$ и инициализацию He, а также регуляризацию $Max - norm$.

Теперь рассмотрим, как проходит процесс обучения модели:

```
batch_size = 256
no_epochs = 10
```

batch_size: Размер батча (пакета) — это количество образцов, которые модель будет одновременно обрабатывать за один шаг обновления весов. В данном случае, за один шаг обучения обрабатывается 256 примеров (окон сигналов).

`no_epochs`: Количество эпох обучения — это число полных проходов модели по всему набору данных. Например, если в данных содержится 1000 образцов, а $batch\_size = 256$, то модель выполнит 4 шага за одну эпоху (так как 1000 / 256 ≈ 4).

```
history = model.fit(data_noisy_flat, data_pure_flat,
                    epochs=no_epochs,
                    batch_size=batch_size,
                    validation_split=validation_split)
```

В результате обучения в 4 эпохи, было получено:

```
Epoch 3/4
211/211 ━━━━━━━━━━━━━━━━━━━━ 5s 26ms/step - loss: 0.4072 - val_loss: 0.4077
Epoch 4/4
211/211 ━━━━━━━━━━━━━━━━━━━━ 5s 26ms/step - loss: 0.4068 - val_loss: 0.4077
```

Рисунок 4.3 – Обучение модели

Что довольно неплохой показатель, увеличении числа эпох или размера выборки, значительно не меняет качество обучения. Сравнение качества удаления шума будет приведено в разделе с проверкой алгоритмов фильтрации.

5 Метрики для проверки алгоритмов фильтрации

Метрики R^2, MSE и MAE являются важными инструментами для оценки качества регрессионных моделей. Они позволяют количественно оценить, насколько хорошо предсказанные значения модели соответствуют фактическим значениям.

Коэффициент детерминации (R^2):

1) R^2 измеряет долю дисперсии зависимой переменной, которая объясняется независимыми переменными в модели. Он принимает значения от 0 до 1, где:
2) $R^2 = 0$: Модель не объясняет никакой дисперсии зависимой переменной.
3) $R^2 = 1$: Модель идеально объясняет всю дисперсию зависимой переменной.

Вычисляется по следующей формуле:

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}} \tag{5.1}$$

где:

SS_res - сумма квадратов остатков регрессии

SS_tot - общая сумма квадратов

$$SS_{res} = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \tag{5.2}$$

$$SS_{res} = \sum_{i=1}^{n} (y_i - \bar{y}_i)^2 \tag{5.3}$$

где:

y – фактическое значение

$\hat{y}_i$ - расчетное значение

R^2 вычисляется путем сравнения суммы квадратов разностей между фактическими и предсказанными значениями (остатки) с суммой квадратов разностей между фактическими значениями и средним значением зависимой переменной. Чем меньше остатки, тем выше R^2.

Среднеквадратичная ошибка (MSE)

MSE измеряет среднюю квадратичную разницу между фактическими и предсказанными значениями. Она чувствительна к выбросам, так как квадратично накапливает ошибки.

Формула MSE:

$$MSE = \left(\frac{1}{n}\right) * \Sigma(y_i - \hat{y}_i)^2 \quad (5.4)$$

где:

n - количество наблюдений

MSE вычисляется путем усреднения квадратов разностей между фактическими и предсказанными значениями. Чем меньше MSE, тем лучше модель.

Средняя абсолютная ошибка (MAE)

MAE измеряет среднюю абсолютную разницу между фактическими и предсказанными значениями. Она менее чувствительна к выбросам, чем MSE.

Формула MAE:

$$MAE = \left(\frac{1}{n}\right) * \Sigma|y_i - \hat{y}_i| \quad (5.5)$$

MAE вычисляется путем усреднения абсолютных разностей между фактическими и предсказанными значениями. Чем меньше MAE, тем лучше модель.

Выбор метрики зависит от конкретной задачи и данных:

1) R^2: Полезен для оценки общей $goodness - of - fit$ модели.
2) MSE: Полезен, когда важны большие ошибки.
3) MAE: Полезен, когда важны все ошибки одинаково.

Реализации алгоритмов можно найти в библиотеке $from\ sklearn.metrics$.

Теперь сравним результаты фильтрации различными алгоритмами.

Для этого сгенерируем несколько сигналов со случайными частотами, наложим на них нормальный и импульсный шум и применим к ним рассмотренные выше алгоритмы фильтрации. Результаты оценим по метрикам из списка выше. Код для сравнений будет приведен в приложении В.

На тепловых картах использовалось две цветовые палитры. Зеленная для значений меньше среднего по таблице, и теплая для тех, что больше.

Рисунок 5.1 – Градиент для значений метрик меньше среднего по всем значениям

Рисунок 5.2 – Градиент для значений метрик выше среднего по всем значениям

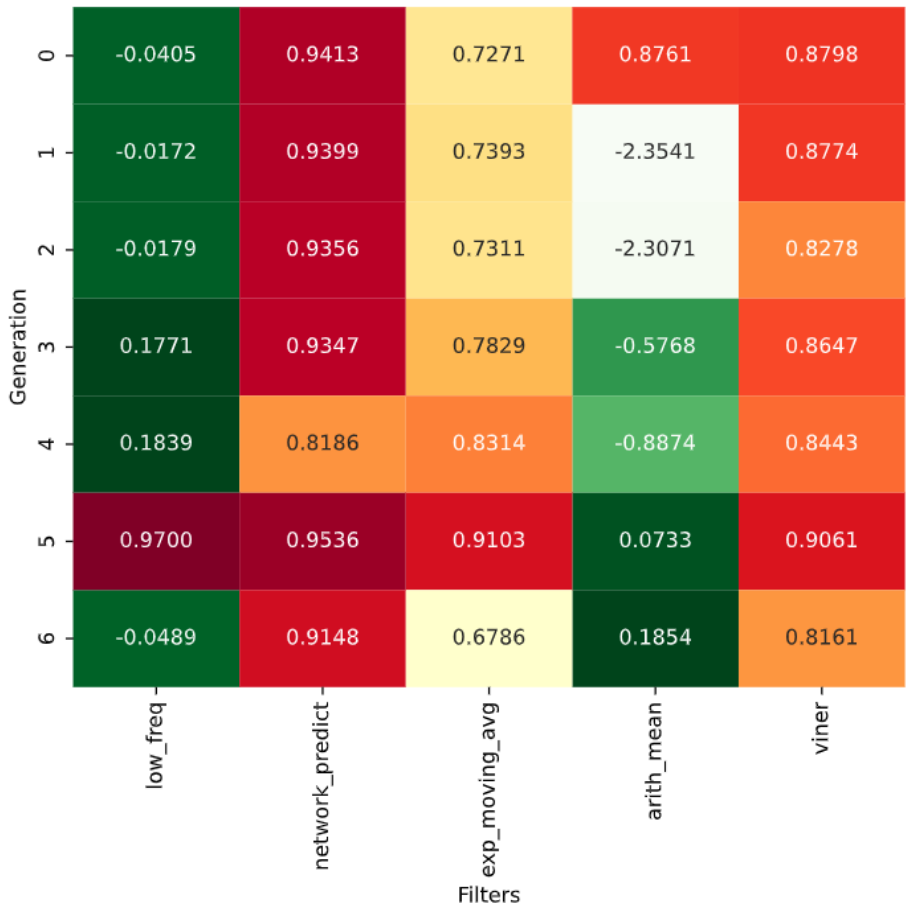

Рисунок 5.2 – Сравнение по метрике R2 алгоритмов фильтрации метрика для 7 случайных сигналов

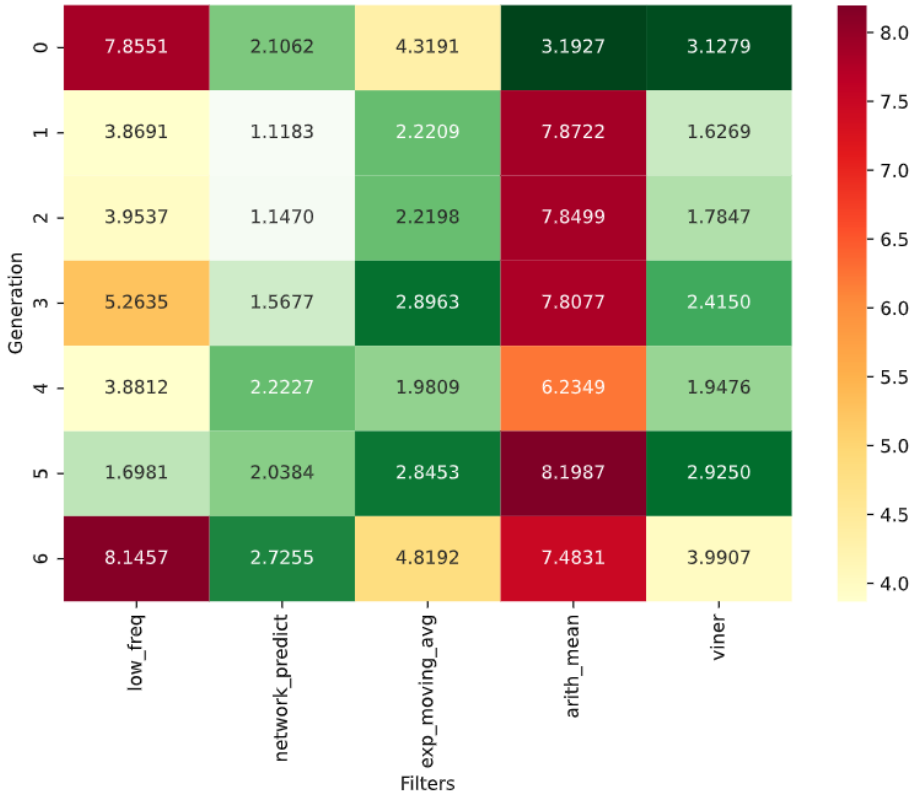

Рисунок 5. 3 – Сравнение по метрике MAE алгоритмов фильтрации метрика для 7 случайных сигналов

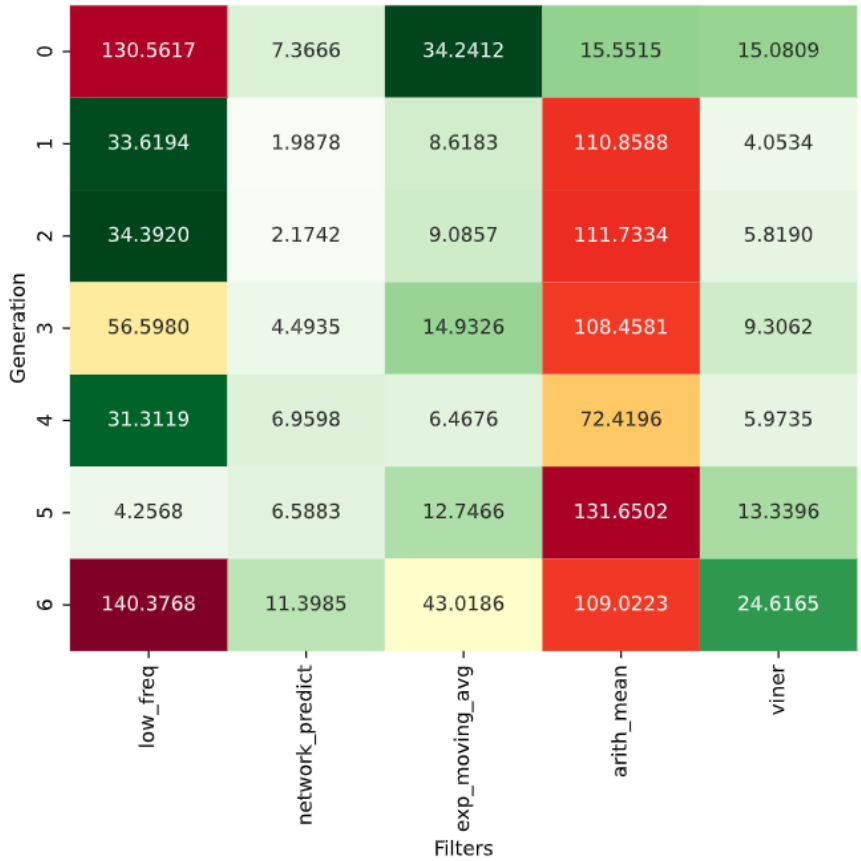

Рисунок 5. 4 – Сравнение по метрике MSE алгоритмов фильтрации метрика для 7 случайных сигналов

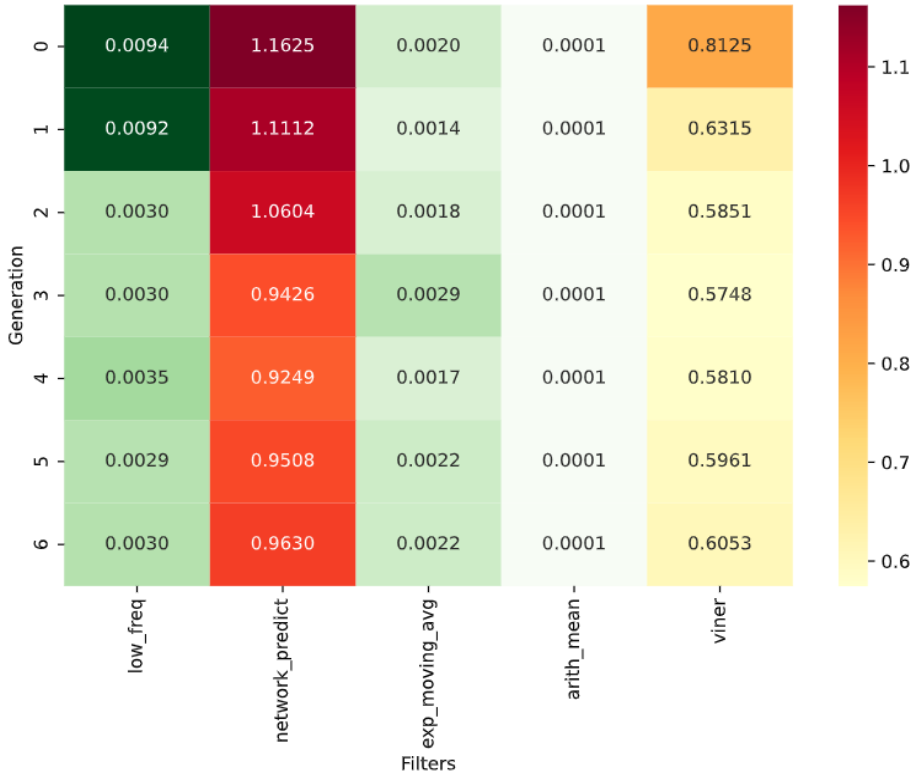

Рисунок 5. 5 – Сравнение по времени выполнения алгоритмов фильтрации метрика для 7 случайных сигналов

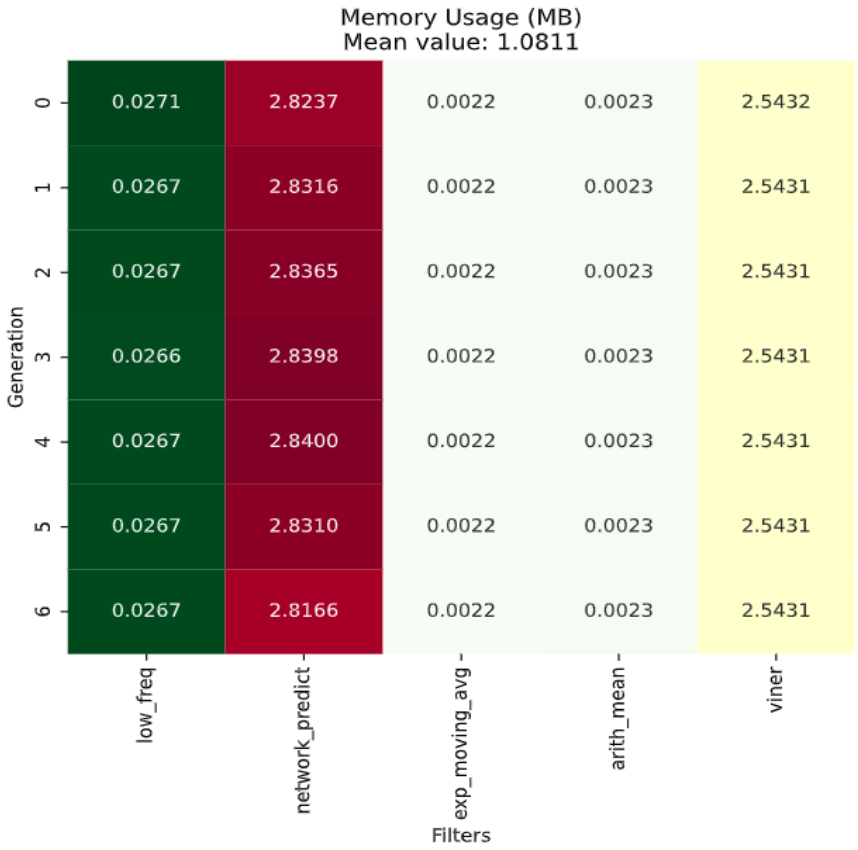

Рисунок 5. 6 – Сравнение по использованию памяти алгоритмами фильтрации метрика для 7 случайных сигналов

Как видно из рисунков выше, фильтрация на основе автоэнкодера дает стабильно лучшие результаты фильтрации, чем прочие алгоритмы, но при этом она требует в среднем в 3 раза больше времени выполнения и ее затраты памяти стабильно велики, так как связаны с загрузкой модели в оперативную память. Так же хорошие результаты получил фильтр Винера, при этом он затрачивает в среднем в два раза меньше времени на фильтрацию, чем автоэнкодер.

6 Заключение

В ходе выполнения курсовой работы были изучены и реализованы различные методы обработки и фильтрации сигналов.

В первом разделе было выполнено моделирование полигармонического сигнала с добавлением шума, что позволило создать тестовые данные для последующего анализа и обработки. Сгенерированный сигнал включал в себя как полезную составляющую, так и шумовую компоненту с гауссовским распределением и небольшой импульсной составляющей.

Во втором разделе был проведен спектральный анализ полученного сигнала, что позволило изучить его частотные характеристики и определить особенности спектрального состава как чистого, так и зашумленного сигнала. Это дало теоретическую основу для реализации фильтра низких частот, а также слоя частотной фильтрации в архитектуре нейросети.

В третьем разделе были реализованы и исследованы традиционные алгоритмы фильтрации: фильтр низких частот, метод скользящего среднего, метод экспоненциального среднего и фильтр Винера.

В четвертом разделе был реализован и исследован современный метод фильтрации на основе автоэнкодера, описана архитектура сети, выбранные для обучения гиперпараметры, а также методы нормировки и предварительной фильтрации.

В пятом разделе были применены различные метрики для оценки качества работы реализованных алгоритмов фильтрации. Были построены тепловые карты по метрикам, а также времени выполнения и затратам памяти. Было показано, что точность нейросетевого подхода стабильно выше других рассмотренных алгоритмов, но затрачиваемые системные ресурсы на фильтрацию, также на порядок больше.

Таким образом, в ходе работы было проведено сравнение алгоритмов, на основе которого можно, в зависимости от требований конкретной задачи, выбрать подходящий алгоритм для задачи фильтрации.

7 Список использованных источников

1.Y. Luo и N. Mesgarani, «Conv-TasNet: Surpassing Ideal Time-Frequency Magnitude Masking for Speech Separation», *IEEEACM Trans. Audio Speech Lang. Process.*, т. 27, вып. 8, сс. 1256–1266, авг. 2019, doi: 10.1109/TASLP.2019.2915167.

2.G. Alain и Y. Bengio, «What Regularized Auto-Encoders Learn from the Data Generating Distribution», 19 август 2014 г., *arXiv*: arXiv:1211.4246. doi: 10.48550/arXiv.1211.4246.

3.A. Eberhardt, Q. Liang, и E. G. M. Ferreira, «de Broglie scale time delays in pulsar networks for ultralight dark matter», 27 ноябрь 2024 г., *arXiv*: arXiv:2411.18051. doi: 10.48550/arXiv.2411.18051.

4.J.-M. Valin, «A Hybrid DSP/Deep Learning Approach to Real-Time Full-Band Speech Enhancement», 31 май 2018 г., *arXiv*: arXiv:1709.08243. doi: 10.48550/arXiv.1709.08243.

5.D. J. Im, M. I. D. Belghazi, и R. Memisevic, «Conservativeness of untied auto-encoders», 21 сентябрь 2015 г., *arXiv*: arXiv:1506.07643. doi: 10.48550/arXiv.1506.07643.

6. Вандер Плас Дж. Python для сложных задач: наука о данных и машинное обучение. — СПб.: Питер, 2018. — 576 с.: ил. — (Серия «Бестселлеры O’Reilly»).

7. Нильсен Э. Практический анализ временных рядов: прогнозирование со статистикой и машинное обучение.: Пер. с англ. — СПб. : ООО “Диалектика”, 2021. — 544 с.: ил. — Парал. тит. англ.

ПРИЛОЖЕНИЕ А

(полный код алгоритма экспоненциального скользящего среднего)

```
def exponential_moving_avereage(data, smooth_interval = 2):
    if smooth_interval >= len(data):
        print("Smooth interval more or equal array lenght!")
        return

    alpha = 2.0/(smooth_interval+1)
    filtered_values = np.zeros(len(data))
    sum = 0

    for step in range(smooth_interval):
        sum += data[step]
    previous_ma_value = sum/smooth_interval

    for step in range(smooth_interval-1):
        previous_ma_value = alpha*data[step]+(1-alpha)*previous_ma_value
        filtered_values[step] = previous_ma_value

    sum = 0

    for step in range(smooth_interval):
        sum += data[step]
    previous_ma_value = sum/smooth_interval
    filtered_values[smooth_interval-1] = previous_ma_value

    for step in range(smooth_interval, len(data)):
        previous_ma_value = alpha*data[step]+(1-alpha)*previous_ma_value
        filtered_values[step] = previous_ma_value

    return filtered_values
```

ПРИЛОЖЕНИЕ Б

(полный код архитектуры автоэнкодера)

```
input_shape = (512, 1)

validation_split = 0.2
verbosity = 1
max_norm_value = 2.0
window_size = N  # Размер окна для разбиения данных
@keras.saving.register_keras_serializable()
class FFTLayer(layers.Layer):
    def __init__(self, inverse=False, **kwargs):
        super(FFTLayer, self).__init__()
        self.inverse = inverse

    def call(self, inputs):
        inputs = tf.cast(inputs, dtype=tf.float32)
        if self.inverse:
            # IFFT
            iftf_result = tf.signal.ifft(tf.complex(inputs, tf.zeros_like(inputs)))
            real_part = tf.math.real(iftf_result)
            return tf.expand_dims(real_part, axis=-1)
        else:
            # FFT
            fft_result = tf.signal.fft(tf.complex(inputs, tf.zeros_like(inputs)))
            real_part = tf.math.real(fft_result)
            return tf.expand_dims(real_part, axis=-1)
@keras.saving.register_keras_serializable()
class LowPassFilter(layers.Layer):
    def __init__(self, threshold=0.7, **kwargs):
        super(LowPassFilter, self).__init__()
        self.threshold = threshold

    def call(self, inputs):
        fft_val = inputs
        mask = fft_val < self.threshold * tf.reduce_mean(fft_val)
        fft2 = tf.where(mask, tf.zeros_like(fft_val), fft_val)
        mask1 = fft_val > self.threshold * tf.reduce_max(fft2)*tf.constant(0.8)
        return tf.where(mask1, tf.zeros_like(fft_val), fft_val)
```

```
input_layer = tf.keras.layers.Input(shape=input_shape, dtype=tf.float32)
fft_layer = FFTLayer(inverse=False)(input_layer)
filter = LowPassFilter()(fft_layer)
# Перетаскивание shape обратно
reshaped_fft = tf.keras.layers.Reshape(input_shape)(filter)
x = tf.keras.layers.Conv1D(128, kernel_size=4,
kernel_constraint=tf.keras.constraints.MaxNorm(max_norm_value), activation='relu',
kernel_initializer='he_uniform', name='conv1')(reshaped_fft)
x = tf.keras.layers.Conv1D(32, kernel_size=4,
kernel_constraint=tf.keras.constraints.MaxNorm(max_norm_value), activation='relu',
kernel_initializer='he_uniform', name='conv2')(x)
x = tf.keras.layers.Conv1DTranspose(32, kernel_size=4,
kernel_constraint=tf.keras.constraints.MaxNorm(max_norm_value), activation='relu',
kernel_initializer='he_uniform', name='convT1')(x)
x = tf.keras.layers.Conv1D(128, kernel_size=4,
kernel_constraint=tf.keras.constraints.MaxNorm(max_norm_value), activation='relu',
kernel_initializer='he_uniform', name='conv3')(x)
x = tf.keras.layers.Conv1DTranspose(128, kernel_size=4,
kernel_constraint=tf.keras.constraints.MaxNorm(max_norm_value), activation='relu',
kernel_initializer='he_uniform', name="convT2")(x)
x = tf.keras.layers.Conv1DTranspose(128, kernel_size=4,
kernel_constraint=tf.keras.constraints.MaxNorm(max_norm_value), activation='relu',
kernel_initializer='he_uniform', name='convT3')(x)
x = tf.keras.layers.Conv1D(1, kernel_size=4,
kernel_constraint=tf.keras.constraints.MaxNorm(max_norm_value), activation='sigmoid',
padding='same', name='conv4')(x)

fft_layer_inv = FFTLayer(inverse=True)(x)
reshaped_input = tf.keras.layers.Reshape(input_shape)(fft_layer_inv)

model = tf.keras.Model(inputs=input_layer, outputs=reshaped_input )
model.summary()



model.compile(optimizer='adam', loss='binary_crossentropy')
```

ПРИЛОЖЕНИЕ В

(полный код для сравнения алгоритмов)

```
N = 256
pd.set_option('display.max_rows', None)
pd.set_option('display.max_columns', None)
pd.set_option('display.width', None)
pd.set_option('display.max_colwidth', None)

def create_metrics_table(signals_count=5):
    # Создаем словари для хранения результатов
    r2_results = {'generation': []}
    mae_results = {'generation': []}
    mse_results = {'generation': []}
    time_results = {'generation': []}
    memory_results = {'generation': []}

    # Названия фильтров
    filters = {
        'low_freq': low_freq,
        'network_predict': network_predict,
        'exp_moving_avg': lambda x: exponential_moving_avereage(x, 2),
        'arith_mean': lambda x: arith_mean(x, 8),
        'viner': viner
    }

    # Инициализируем колонки для каждого фильтра
    for filter_name in filters.keys():
        r2_results[filter_name] = []
        mae_results[filter_name] = []
        mse_results[filter_name] = []
        time_results[filter_name] = []
        memory_results[filter_name] = []

    last_signals = {}
    # Генерируем сигналы и применяем фильтры
    for i in range(signals_count):
        clean_signal, noisy_signal = get_signal(N)  # предполагаем, что длина
сигнала 1000
```

```
            r2_results['generation'].append(i+1)
            mae_results['generation'].append(i+1)
            mse_results['generation'].append(i+1)
            time_results['generation'].append(i+1)
            memory_results['generation'].append(i+1)

            if i == signals_count - 1:
                last_signals['clean'] = clean_signal
                last_signals['noisy'] = noisy_signal


            # Применяем каждый фильтр
            for filter_name, filter_func in filters.items():
                # Замеряем время выполнения
                start_time = time.time()
                filtered_signal, memory_used = get_memory_usage(filter_func,
noisy_signal)
                end_time = time.time()

                if i == signals_count - 1:
                    last_signals[filter_name] = filtered_signal


                # Вычисляем метрики
                r2 = r2_score(clean_signal, filtered_signal)
                mae = mean_absolute_error(clean_signal, filtered_signal)
                mse = mean_squared_error(clean_signal, filtered_signal)

                r2 = np.round(r2_score(clean_signal, filtered_signal), decimals=4)
                mae = np.round(mean_absolute_error(clean_signal, filtered_signal),
decimals=4)
                mse = np.round(mean_squared_error(clean_signal, filtered_signal),
decimals=4)
                exec_time = np.round(end_time - start_time, decimals=4)

                # Сохраняем результаты
                r2_results[filter_name].append(r2)
                mae_results[filter_name].append(mae)
                mse_results[filter_name].append(mse)
```

```
            time_results[filter_name].append(exec_time)
            memory_results[filter_name].append(memory_used)


        # Создаем DataFrame для каждой метрики
        r2_df = pd.DataFrame(r2_results)
        mae_df = pd.DataFrame(mae_results)
        mse_df = pd.DataFrame(mse_results)
        time_df = pd.DataFrame(time_results)
        memory_df = pd.DataFrame(memory_results)

        # Сохраняем результаты в CSV
        r2_df.to_csv('r2_scores.csv', index=False)
        mae_df.to_csv('mae_scores.csv', index=False)
        mse_df.to_csv('mse_scores.csv', index=False)
        time_df.to_csv('execution_times.csv', index=False)
        memory_df.to_csv('memory_usage.csv', index=False)

         # Визуализация результатов последнего сигнала
        plt.figure(figsize=(15, 10))

        # График исходного и зашумленного сигнала
        plt.subplot(2, 1, 1)
        plt.plot(last_signals['clean'], label='Clean Signal', alpha=0.7)
        plt.plot(last_signals['noisy'], label='Noisy Signal', alpha=0.5)
        plt.title('Original and Noisy Signals')
        plt.legend()
        plt.grid(True)

        # График результатов фильтрации
        plt.subplot(2, 1, 2)
        plt.plot(last_signals['clean'], label='Clean Signal', alpha=0.7)
        print(filters.keys())
        for filter_name in filters.keys():
            plt.plot(last_signals[filter_name], label=f'Filtered ({filter_name})',
alpha=0.5)
        plt.title('Filtering Results')
        plt.legend()
        plt.grid(True)
```

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

    # Визуализация метрик в виде тепловых карт
    metrics_dfs = {
        'R2 Score': r2_df.drop('generation', axis=1),
        'MAE': mae_df.drop('generation', axis=1),
        'MSE': mse_df.drop('generation', axis=1),
        'Execution Time': time_df.drop('generation', axis=1),
        'Memory Usage (MB)': memory_df.drop('generation', axis=1)
    }

    fig, axes = plt.subplots(3, 2, figsize=(15, 20))
    fig.suptitle('Metrics Heatmaps', fontsize=16)

    for (title, df), ax in zip(metrics_dfs.items(), axes.flat):
        create_custom_heatmap(df, ax, title)



    return r2_df, mae_df, mse_df, time_df, memory_df

# Запускаем анализ
r2_df, mae_df, mse_df, time_df, memory_df  = create_metrics_table(7)

# Выводим результаты
print("R2 Scores:")
print(r2_df)
print("\nMAE Scores:")
print(mae_df)
print("\nMSE Scores:")
print(mse_df)
print("\nExecution Times:")
print(time_df)
```