МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ

Федеральное государственное бюджетное образовательное учреждение высшего образования «Южно-Российский государственный политехнический университет (НПИ) имени М.И. Платова»

Факультет **информационных технологий и управления**

Кафедра **«Программное обеспечение вычислительной техники»**

Направление **09.04.01 – Информатика и вычислительная техника**

ОТЧЕТ

по Лабораторной работе №2

по дисциплине: **Программное и аппаратное обеспечение информационных систем**

Выполнил студент 1 **курса, группы** ТИСа-о24 Якимов П.В.
Фамилия, имя, отчество

Принял доцент, кандидат технических наук Рыбалкин А.Д.
Фамилия, имя, отчество

«_____» _____2024 г. _____
Подпись

Новочеркасск, 2024 г

Лабораторная работа №2

«Разработка программной части ИС и АС»

Цель работы: Изучить методы сопровождения исходного кода, создания и использования репозиториев методы получения сигналов и данных, методы предварительной обработки информации перед применением бизнес-логики приложения.

Теоретический материал: Современные методы разработки программного обеспечения включают нативные и кроссплатформенные инструменты. Нативные инструменты, такие как C++ и Java, предназначены для создания приложений на определённых платформах. Кроссплатформенные решения, например, JavaScript и Python, позволяют разрабатывать программы, работающие на разных операционных системах.

Системы контроля версий (VCS), такие как Git, помогают управлять изменениями в коде и совместной работой в команде. Основные команды Git, такие как git init, git add, git commit, git push, позволяют организовать процесс разработки и отслеживать историю изменений. Платформы вроде GitHub и GitLab предоставляют инструменты для управления проектами и документации. Предварительная обработка данных включает методы фильтрации и сглаживания для повышения качества данных. Алгоритмы, такие как скользящие средние и экспоненциальное сглаживание (EMA), помогают устранять шумы и улучшать анализ сигналов.

Использование аппаратных средств, таких как Arduino и Raspberry Pi, позволяет получать данные от датчиков и тестировать алгоритмы обработки сигналов. При отсутствии физического оборудования возможно использование математических моделей, файлов или внешних API.

UML и моделирование служат для визуального представления программных систем. UML позволяет создавать диаграммы классов,

последовательностей и потоков данных, что помогает разработчикам лучше понимать структуру и поведение системы.

Ход работы:

1) Разработана блок-схема, демонстрирующая последовательность работы программы. Схема включает ключевые этапы: определение функций и класса SignalApp, а также обработку нажатий кнопок "Старт" и "Стоп", что иллюстрирует общую логику приложения, как показано на рисунке 1.

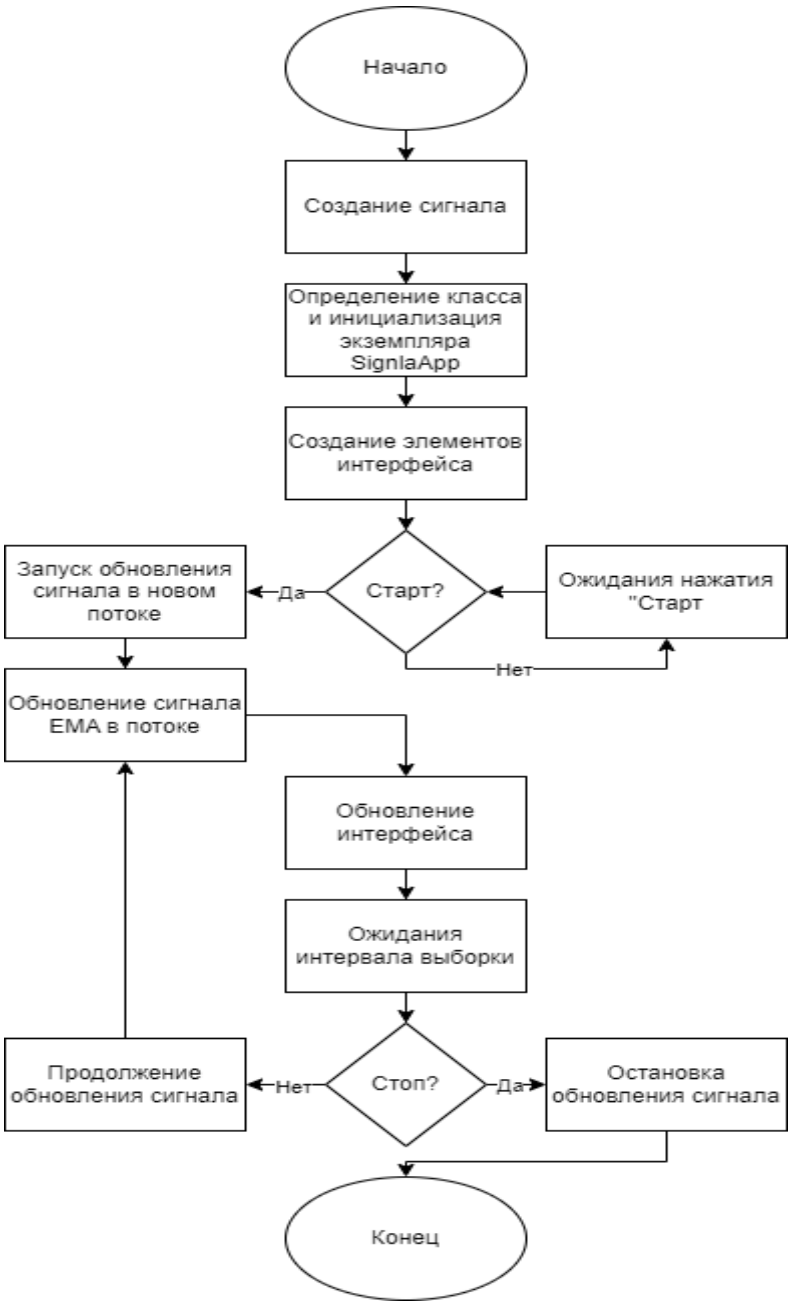

Рисунок 1 – Блок-схема программы

2) В первой части кода происходит импорт необходимых библиотек, таких как NumPy, для числовых операций, модули для работы с временем и потоками, а также библиотеки для создания графического интерфейса и построения графиков. Основной функцией является *generate_signal_single_point*, которая принимает параметры, включая временную точку и количество членов для различных компонентов сигнала. Внутри функции инициализируются константы для экспоненциального затухания, базовой частоты косинусных членов и логарифмических компонентов, а затем происходит суммирование сигналов, что иллюстрируется на рисунке 2.

```python
# Функция для генерации сигнала в один момент времени
GigaCode: explain | explain step by step | doc | test
def generate_signal_single_point(time_point, num_exp, num_cos, num_log, amp_exp, amp_cos, amp_log):
    total_signal = 0

    # Константы для компонентов сигнала
    exp_const = 1
    freq_base = 2 * np.pi
    phase_shift = 0
    log_const = 1
    k_base = 1

    # Экспоненциальные компоненты
    if num_exp > 0:
        for i in range(num_exp):
            exp_amp = amp_exp[i] if i < len(amp_exp) else 0
            total_signal += exp_amp * np.exp(-time_point / exp_const)

    # Косинусные компоненты
    if num_cos > 0:
        for j in range(num_cos):
            cos_amp = amp_cos[j] if j < len(amp_cos) else 0
            frequency = freq_base * (j + 1)
            total_signal -= cos_amp * np.cos(frequency * time_point + phase_shift)

    # Логарифмические компоненты
    if num_log > 0:
        for k in range(num_log):
            log_amp = amp_log[k] if k < len(amp_log) else 0
            k_value = k_base * (k + 1)
            # Избегаем логарифма нуля или отрицательных чисел
            log_input = max(log_const * k_value * time_point, 1e-10)
            total_signal += log_amp * np.log10(log_input)

    return total_signal
```

Рисунок 2 – Создание функции для генерации сигнала

3) Далее пределяется класс SignalApp, который отвечает за создание графического интерфейса для генерации и обработки сигнала. Внутри конструктора инициализируются параметры для генерации сигнала, а также массивы амплитуд для каждого типа сигнала. Задаются параметры для расчета экспоненциального скользящего среднего, включая сглаживающий фактор и массивы для хранения данных. В конце инициализации вызывается метод generate_widgets, задающий элементы интерфейса, что показано на рисунке 3.

```
# Класс для графического интерфейса
GigaCode: explain | explain step by step | doc | test
class SignalApp:
    def __init__(self, master):
        self.master = master
        self.master.title("Генерация и обработка сигнала")

        # Параметры для генерации сигнала
        self.num_exp = 3
        self.num_cos = 1
        self.num_log = 0
        self.amp_exp = [0.32, 0.15, 0.41]
        self.amp_cos = [1.2]
        self.amp_log = [0]

        # Параметры для расчета EMA
        self.N = 5
        self.alpha = 2 / (self.N + 1)
        self.EMA_prev = None

        # Массивы для хранения данных
        self.time_values = []
        self.signal_values = []
        self.ema_values = []

        # Параметры времени
        self.sampling_interval = 0.1
        self.total_duration = 30
        self.start_time = None
        self.current_time = 0
        self.running = False

        # Создаем элементы интерфейса
```

Рисунок 3 – Определение класса SignalApp

4) Метод generate_widgets отвечает за создание и размещение элементов управления в графическом интерфейсе. В нем создаются кнопки для начала и остановки генерации сигнала, а также поля для отображения текущих значений времени, сигнала и EMA. Каждое поле подписано, чтобы можно было легко следить за изменениями значений. Эти элементы расположены в соответствующих фреймах, обеспечивая структурированный и удобный интерфейс. Всё это проиллюстрировано на рисунке 4.

```
def generate_widgets(self):
    # Кнопки управления
    control_frame = ttk.Frame(self.master)
    control_frame.pack(side=tk.TOP, fill=tk.X)

    self.start_button = ttk.Button(control_frame, text="Старт", command=self.start_signal)
    self.start_button.pack(side=tk.LEFT, padx=5, pady=5)

    self.stop_button = ttk.Button(control_frame, text="Стоп", command=self.stop_signal, state=tk.DISABLED)
    self.stop_button.pack(side=tk.LEFT, padx=5, pady=5)

    # Поля для отображения текущих значений
    value_frame = ttk.Frame(self.master)
    value_frame.pack(side=tk.TOP, fill=tk.X)

    ttk.Label(value_frame, text="Текущее время:").grid(row=0, column=0, sticky=tk.W, padx=5)
    self.time_label = ttk.Label(value_frame, text="0.00 с")
    self.time_label.grid(row=0, column=1, sticky=tk.W, padx=5)

    ttk.Label(value_frame, text="Текущий сигнал:").grid(row=1, column=0, sticky=tk.W, padx=5)
    self.signal_label = ttk.Label(value_frame, text="0.0000")
    self.signal_label.grid(row=1, column=1, sticky=tk.W, padx=5)

    ttk.Label(value_frame, text="Текущий EMA:").grid(row=2, column=0, sticky=tk.W, padx=5)
    self.ema_label = ttk.Label(value_frame, text="0.0000")
    self.ema_label.grid(row=2, column=1, sticky=tk.W, padx=5)

    # График
    self.figure, self.ax = plt.subplots(figsize=(8, 4))
    self.line1, = self.ax.plot([], [], label='Сигнал')
    self.line2, = self.ax.plot([], [], label='EMA', linestyle='--')
    self.ax.set_xlabel('Время (с)')
    self.ax.set_ylabel('Значение')
    self.ax.set_title('Сигнал и EMA')
    self.ax.legend()
    self.ax.grid(True)

    self.canvas = FigureCanvasTkAgg(self.figure, master=self.master)
    self.canvas.get_tk_widget().pack(side=tk.TOP, fill=tk.BOTH, expand=1)
```

Рисунок 4 – Создание элементов интерфейса

5) В данной части кода создается график для визуализации сигнала и EMA. Используя Matplotlib, создается фигура и оси, на которых будут

отображаться данные. Устанавливаются метки для осей и заголовок графика, а также создаются линии для сигнала и EMA. Затем график упаковывается в интерфейс, что позволяет видеть визуализацию сигналов в реальном времени, что демонстрирует рисунок 5.

```
# График
self.figure, self.ax = plt.subplots(figsize=(8, 4))
self.line1, = self.ax.plot([], [], label='Сигнал')
self.line2, = self.ax.plot([], [], label='EMA', linestyle='--')
self.ax.set_xlabel('Время (с)')
self.ax.set_ylabel('Значение')
self.ax.set_title('Сигнал и EMA')
self.ax.legend()
self.ax.grid(True)
```

Рисунок 5 – Настройка графика для визуализации данных

6) Методы start_signal и stop_signal управляют процессом генерации сигнала. При запуске метода start_signal проверяется, не выполняется ли уже генерация. Если нет, происходит инициализация необходимых параметров, таких как время и массивы для хранения данных. Затем запускается отдельный поток, который будет обновлять данные сигнала. Метод stop_signal останавливает генерацию сигнала. Все действия показаны на рисунке 6.

```
def start_signal(self):
    if not self.running:
        self.running = True
        self.start_button.config(state=tk.DISABLED)
        self.stop_button.config(state=tk.NORMAL)
        self.start_time = time.time()
        self.EMA_prev = None
        self.time_values.clear()
        self.signal_values.clear()
        self.ema_values.clear()
        self.ax.clear()
        self.line1, = self.ax.plot([], [], label='Сигнал')
        self.line2, = self.ax.plot([], [], label='EMA', linestyle='--')
        self.ax.set_xlabel('Время (с)')
        self.ax.set_ylabel('Значение')
        self.ax.set_title('Сигнал и EMA')
        self.ax.legend()
        self.ax.grid(True)
        threading.Thread(target=self.update_signal).start()
```

Рисунок 6 – Управление процессом генерации сигнала

7) В методе update_signal осуществляется непрерывное обновление сигнала, пока работает генерация. В этом методе вычисляется текущее время и вызывается функция generate_signal_single_point для генерации сигнала в данный момент. Также рассчитывается EMA на основе текущего значения сигнала и предыдущего значения EMA, если оно доступно. Это позволяет отслеживать изменения сигнала в реальном времени и показано на рисунке 7.

```
def update_signal(self):
    while self.running and self.current_time <= self.total_duration:
        # Вычисляем текущее время
        self.current_time = time.time() - self.start_time

        # Генерируем сигнал в текущий момент времени
        signal_value = generate_signal_single_point(
            self.current_time,
            self.num_exp,
            self.num_cos,
            self.num_log,
            self.amp_exp,
            self.amp_cos,
            self.amp_log
        )

        # Расчет EMA
        P = signal_value  # Текущее значение сигнала
        if self.EMA_prev is None:
            EMA = P  # Инициализируем EMA первым значением сигнала
        else:
            EMA = (P * self.alpha) + (self.EMA_prev * (1 - self.alpha))

        # Обновляем предыдущее значение EMA
        self.EMA_prev = EMA
```

Рисунок 7 – Обновление сигнала в реальном времени

8) В последней части кода сохранение данных, таких как текущее время, значение сигнала и EMA, в соответствующие массивы. После этого обновляются метки в интерфейсе с текущими значениями, а также обновляется график, чтобы отобразить новые данные. График пересчитывается и перерисовывается, чтобы отражать изменения в сигнале и EMA. Наконец, происходит ожидание следующего интервала выборки, что обеспечивает плавную работу приложения. Происходит запуск приложения, где создается корневое окно с помощью Tkinter и инициализируется класс SignalApp. После этого запускается главный цикл обработки событий интерфейса, что позволяет взаимодействовать с графическим интерфейсом и наблюдать за изменениями сигналов в реальном времени.

```
            # Сохраняем данные
            self.time_values.append(self.current_time)
            self.signal_values.append(signal_value)
            self.ema_values.append(EMA)

            # Обновляем интерфейс
            self.time_label.config(text=f"{self.current_time:.2f} с")
            self.signal_label.config(text=f"{signal_value:.4f}")
            self.ema_label.config(text=f"{EMA:.4f}")

            # Обновляем график
            self.line1.set_data(self.time_values, self.signal_values)
            self.line2.set_data(self.time_values, self.ema_values)
            self.ax.relim()
            self.ax.autoscale_view()
            self.canvas.draw()

            # Ждем следующего интервала выборки
            time.sleep(self.sampling_interval)

        self.running = False
        self.start_button.config(state=tk.NORMAL)
        self.stop_button.config(state=tk.DISABLED)
```

Рисунок 8 – Сохранение данных, обновление интерфейса и запуск

9) На этом этапе проект загружается на платформу для размещения репозиториев, GitHub. Сначала инициализируется репозиторий с помощью команды *git init*, затем добавляются все файлы проекта с помощью команды *git add .*. После этого выполняется коммит изменений с описанием, например, *git commit -m "First commit"*. Для подключения к удаленному репозиторию используется команда *git remote add origin*, после чего проект загружается на удалённый сервер командой *git push -u origin master*. В последствие, проект переносится на GitHub. Весь процесс и результат показаны на рисунке 9.

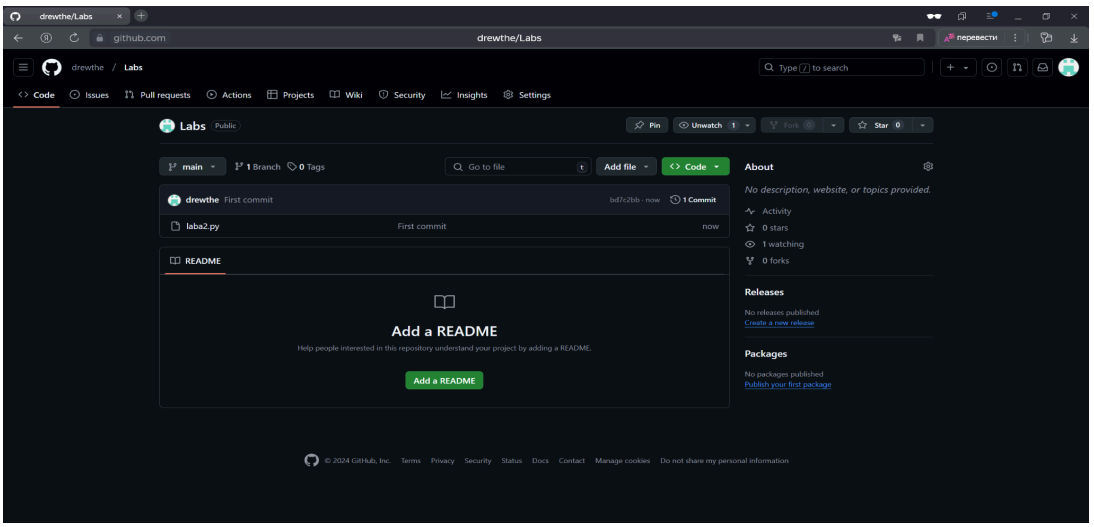

Рисунок 8 – Перенос проекта в GitHub

Результаты работы:

После нажатия кнопки "Старт" происходит генерация сигнала с использованием функции generate_signal_single_point, который состоит из экспоненциальных, косинусных и логарифмических компонентов. График отображает зависимость сигнала от времени: по оси X — время, по оси Y — значение сигнала. Основной сигнал представлен сплошной линией, а скользящее среднее (EMA) — пунктирной. Заголовок графика и подписи осей задаются соответствующими методами Matplotlib. В реальном времени обновляются текущие значения времени, сигнала и EMA, позволяя наблюдать за динамикой сигнала на протяжении всего периода генерации (рисунок 10).

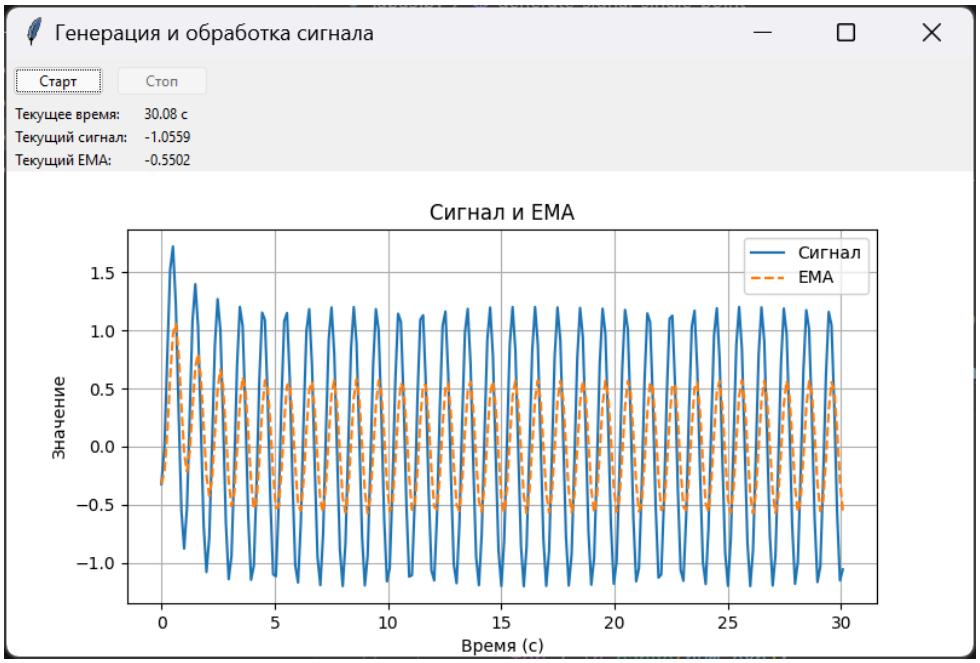

Рисунок 9 – Визуализированный сигнал

Вывод: Разработана программа для генерации и визуализации сложных сигналов. Она демонстрирует методы обработки данных в реальном времени и расчет скользящего среднего (EMA). Графический интерфейс обеспечивает удобное взаимодействие с пользователем, а результаты подтверждают практическую применимость приложения для анализа сигналов.