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 №3

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 №3

«Разработка аппаратной части ИС и АС или ее модели»

Цель работы: Изучить способы управления версиями программного кода, разработать каркас приложений, научиться работать с цифровыми данными, применять алгоритмы предварительной обработки информации, подготавливая для дальнейшего применения, применять CASE-средства поддержки процесса разработки программного обеспечения.

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.

Современные методы разработки программного обеспечения используют нативные и кроссплатформенные инструменты. Нативные инструменты, такие как C++, C#, Java, применяются для создания приложений под конкретные платформы, обеспечивая высокую производительность и полное использование возможностей системы. Кроссплатформенные инструменты, такие как Python и JavaScript, позволяют разрабатывать приложения, работающие на разных операционных системах, что облегчает их перенос и поддержку.

Предварительная обработка данных включает методы фильтрации и сглаживания, которые помогают улучшить качество данных. Алгоритмы, такие как скользящее среднее и экспоненциальное сглаживание (EMA), устраняют шумы и делают сигнал более пригодным для анализа. Эти методы широко применяются при работе с цифровыми сигналами и помогают обеспечить более точные результаты на этапе последующего анализа.

Ход работы:

1) Разработана схема последовательности, которая описывает пошаговый процесс взаимодействия компонентов системы после нажатия пользователем кнопки "Старт". На схеме показано, как SignalApp инициирует процесс генерации сигнала, запуская его в отдельном потоке (Thread). Поток вызывает метод get_signal_from_source() для получения новых данных сигнала. Для этого SignalApp обращается к классу SignalSource, который отвечает за предоставление данных сигнала на основе источника. После получения сигнала в программе выполняются расчёты EMA и скользящего среднего. Затем интерфейс обновляется для отображения текущих данных, включая сигнал, EMA и скользящее среднее. Этот процесс повторяется до тех пор, пока пользователь не нажмёт кнопку "Стоп", что завершает работу приложения. Всё это показано на рисунке 1.

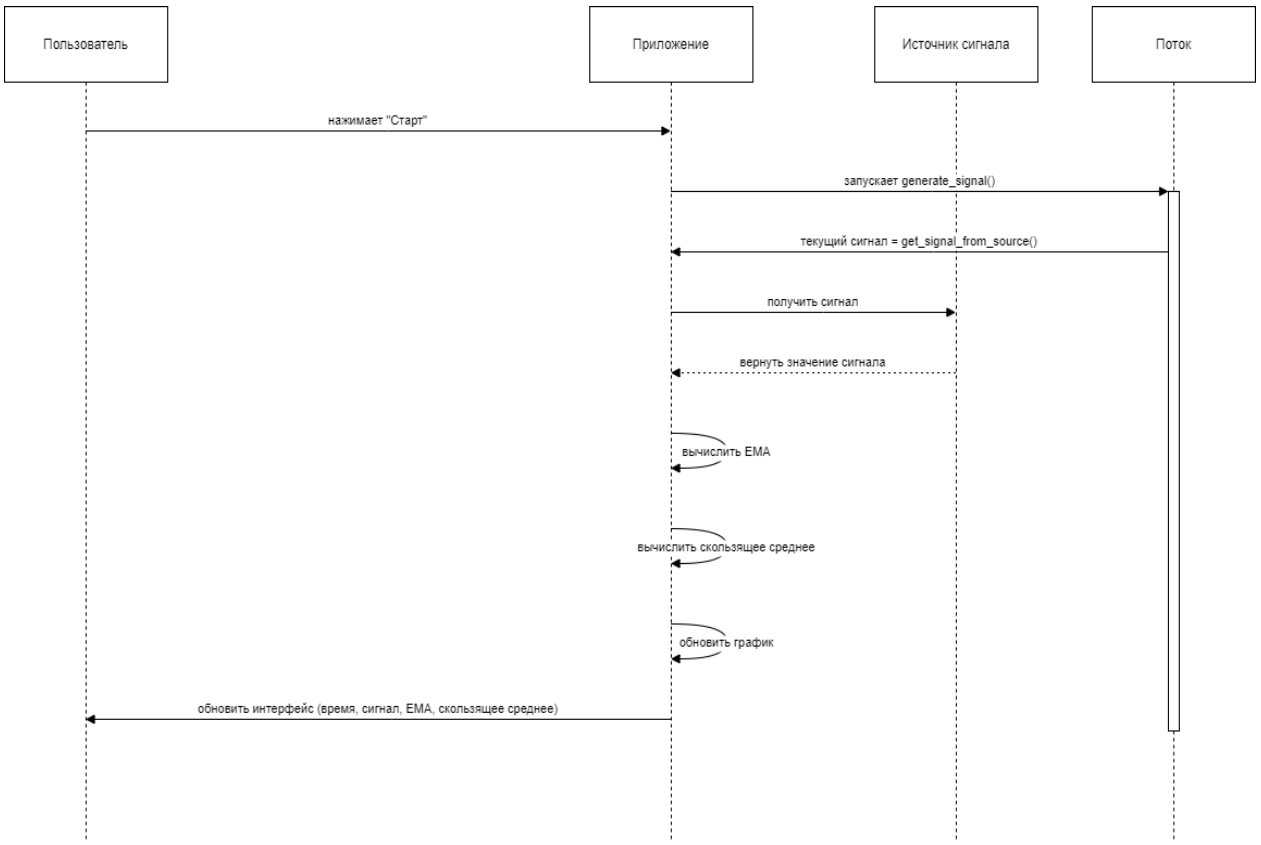

Рисунок 1 – Схема последовательности

2) В первой части кода, помимо импорта необходимых библиотек, таких как NumPy, добавляется библиотека для работы с API, которая может

использоваться для получения сигнала из внешнего источника. Основной функцией является generate_signal_single_point, которая принимает параметры, включая временную точку и количество членов для различных компонентов сигнала. Функция суммирует значения этих компонентов для получения итогового сигнала, что показано 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):

    #Константы
    total_signal = 0
    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)
            log_input = max(log_const * k_value * time_point, 1e-10)
            total_signal += log_amp * np.log10(log_input)

    return total_signal
```

Рисунок 2 – Импорт библиотек и создание функции генерации сигнала

3) В этой части кода создаётся класс SignalApp, который отвечает за создание графического интерфейса для генерации и обработки сигнала. В конструкторе класса инициализируются параметры для генерации сигнала, такие как количество экспоненциальных, косинусных и логарифмических членов, а также их амплитуды. Задаются параметры для расчёта EMA (экспоненциального скользящего среднего) и скользящего среднего. Инициализируются массивы для хранения значений времени, сигнала, EMA и скользящего среднего. Также задаётся источник данных, который может быть либо математической моделью, либо внешним сенсором или API (рисунок 3).

```python
# 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

        # Параметры скользящего среднего
        self.window_size = 3
        self.moving_avg_window = []

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

Рисунок 3 – Создание класса SignalApp и инициализация параметров

4) Метод generate_widgets отвечает за создание и размещение элементов управления в графическом интерфейсе. В нём создаются кнопки для управления процессом генерации сигнала («Старт» и «Стоп»), а также поля для отображения текущих значений времени, сигнала, EMA и скользящего среднего. Структурированный интерфейс позволяет управлять процессом генерации и наблюдать за изменениями в данных в режиме реального времени, что продемонстрировано на рисунке 4.

```
# Параметры времени
self.sampling_interval = 0.05
self.total_duration = 10
self.start_time = None
self.current_time = 0
self.running = False

# Создаем элементы интерфейса
self.generate_widgets()

# Инициализация источника данных
self.data_source = "mathematical_model"
```

Рисунок 4 – Создание элементов управления и отображения интерфейса

5) Здесь создаётся график для визуализации данных. Используя библиотеку Matplotlib, создаётся фигура и оси для графика, на которых будут отображаться линии для сигнала, EMA и скользящего среднего. Устанавливаются метки осей («Время» и «Значение») и заголовок графика. Линии для сигнала, EMA и скользящего среднего создаются с различными стилями (сплошная, пунктирная и точечная линии). Этот график будет обновляться в реальном времени во время генерации сигнала (рисунок 5).

```python
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)

    ttk.Label(value_frame, text="Скользящее среднее:").grid(row=3, column=0, sticky=tk.W, padx=5)
    self.moving_avg_label = ttk.Label(value_frame, text="0.0000")
    self.moving_avg_label.grid(row=3, 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.line3, = self.ax.plot([], [], label='Скользящее среднее', linestyle=':')
    self.ax.set_xlabel('Время (с)')
    self.ax.set_ylabel('Значение')
    self.ax.set_title('Сигнал, EMA и Скользящее Среднее')
    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 5 – Настройка графика для визуализации сигнала и EMA

6) Методы start_signal и stop_signal управляют процессом генерации сигнала. Метод start_signal запускает генерацию, инициализируя параметры времени, обнуляя массивы данных и создавая отдельный поток для генерации сигнала. Также происходит обновление интерфейса для отображения нового сигнала. Метод stop_signal останавливает процесс генерации, деактивируя поток и меняя состояние кнопок управления, что показано 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.moving_avg_values.clear()
        self.ax.clear()
        self.line1, = self.ax.plot([], [], label='Сигнал')
        self.line2, = self.ax.plot([], [], label='EMA', linestyle='--')
        self.line3, = self.ax.plot([], [], label='Скользящее среднее', linestyle=':')
        self.ax.set_xlabel('Время (с)')
        self.ax.set_ylabel('Значение')
        self.ax.set_title('Сигнал, EMA и Скользящее Среднее')
        self.ax.legend()
        self.ax.grid(True)

        # Запуск потока для генерации сигнала
        self.signal_thread = threading.Thread(target=self.generate_signal)
```

Рисунок 6 – Управление процессом генерации сигнала

7) Метод get_signal_from_source отвечает за выбор источника данных для сигнала. В зависимости от выбранного источника (математическая модель, аналоговый или цифровой сенсор, либо API), метод возвращает соответствующее значение сигнала. Например, для математической модели используется функция generate_signal_single_point, для сенсоров — имитация случайных данных, а для API — запрос на сервер для получения сигнала, что показано на рисунке 7.

```
def stop_signal(self):
    if self.running:
        self.running = False
        self.start_button.config(state=tk.NORMAL)
        self.stop_button.config(state=tk.DISABLED)

def get_signal_from_source(self):
    if self.data_source == "mathematical_model":
        return 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)
    elif self.data_source == "analog_sensor":
        # Имитация данных от аналогового датчика
        return np.random.rand()
    elif self.data_source == "digital_sensor":
        # Имитация данных от цифрового датчика
        return np.random.randint(0, 256)
    elif self.data_source == "api":
        try:
            response = requests.get('https://api.example.com/signal')
            return response.json().get('signal', 0)
        except Exception as e:
            print(f"Ошибка при получении данных из API: {e}")
            return 0
```

Рисунок 7 – Получение данных сигнала из различных источников

8) Метод generate_signal отвечает за непрерывную генерацию сигнала в реальном времени. Во время работы этого метода вычисляется текущее время, получаются новые значения сигнала, вычисляются EMA и скользящее среднее. Значения сохраняются в массивы и используются для обновления графика, отображаемого на экране. Также обновляются текстовые метки, показывающие текущие значения сигнала, EMA и скользящего среднего, что продемонстрировано на рисунке 8.

```
def generate_signal(self):
    while self.running and self.current_time < self.total_duration:
        self.current_time = time.time() - self.start_time
        signal_value = self.get_signal_from_source()

        # Расчет EMA
        if self.EMA_prev is None:
            self.EMA_prev = signal_value
        else:
            self.EMA_prev = (self.alpha * signal_value) + ((1 - self.alpha) * self.EMA_prev)

        # Расчет скользящего среднего
        self.moving_avg_window.append(signal_value)
        if len(self.moving_avg_window) > self.window_size:
            self.moving_avg_window.pop(0)
        moving_avg = sum(self.moving_avg_window) / len(self.moving_avg_window)

        # Сохраняем значения для построения графиков
        self.time_values.append(self.current_time)
        self.signal_values.append(signal_value)
        self.ema_values.append(self.EMA_prev)
        self.moving_avg_values.append(moving_avg)

        # Обновляем линии на графике
        self.line1.set_data(self.time_values, self.signal_values)
        self.line2.set_data(self.time_values, self.ema_values)
        self.line3.set_data(self.time_values, self.moving_avg_values)
```

Рисунок 8 – Генерация сигнала и обновление графика в реальном времени

9) В последней части задаются динамические пределы осей для графика в зависимости от текущих значений сигнала, EMA и скользящего среднего. Это предотвращает ситуацию, когда график становится неподвижным из-за одинаковых значений по оси Y. Также обновляется визуальное отображение графика с помощью метода canvas.draw, который перерисовывает содержимое. Всё это показано на рисунке 9.

```
# Устанавливаем пределы осей
self.ax.set_xlim(0, max(self.time_values) if self.time_values else 1)

# Избегаем одинаковых значений границ по оси Y
min_value = min(min(self.signal_values), min(self.ema_values), min(self.moving_avg_values))
max_value = max(max(self.signal_values), max(self.ema_values), max(self.moving_avg_values))

if min_value == max_value:
    min_value -= 0.1
    max_value += 0.1

self.ax.set_ylim(min_value, max_value)

# Обновляем интерфейс
self.canvas.draw()

# Обновляем текстовые метки
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"{self.EMA_prev:.4f}")
self.moving_avg_label.config(text=f"{moving_avg:.4f}")

time.sleep(self.sampling_interval)
```

Рисунок 9 – Обновление пределов осей и перерисовка графика

Результаты работы:

После нажатия кнопки "Старт" происходит генерация сигнала с использованием функции generate_signal_single_point, которая создает сигнал, состоящий из экспоненциальных, косинусных и логарифмических компонентов. В отдельном потоке сигнал обновляется в реальном времени с заданным интервалом. На графике, отображаемом в интерфейсе, по оси X показывается время, а по оси Y — значения сигнала. Основной сигнал представлен сплошной линией, EMA — пунктирной, а скользящее среднее — точечной линией. В реальном времени обновляются текущие значения

времени, сигнала, EMA и скользящего среднего, которые отображаются в соответствующих текстовых полях интерфейса. График автоматически обновляется с учётом текущих данных, что позволяет пользователю наблюдать за динамикой изменения сигнала. Пользователь может остановить генерацию сигнала в любой момент, нажав кнопку "Стоп", при этом поток прекращает свою работу. Результаты отображения динамики сигнала и его обработки показаны на рисунках 10 и 11.

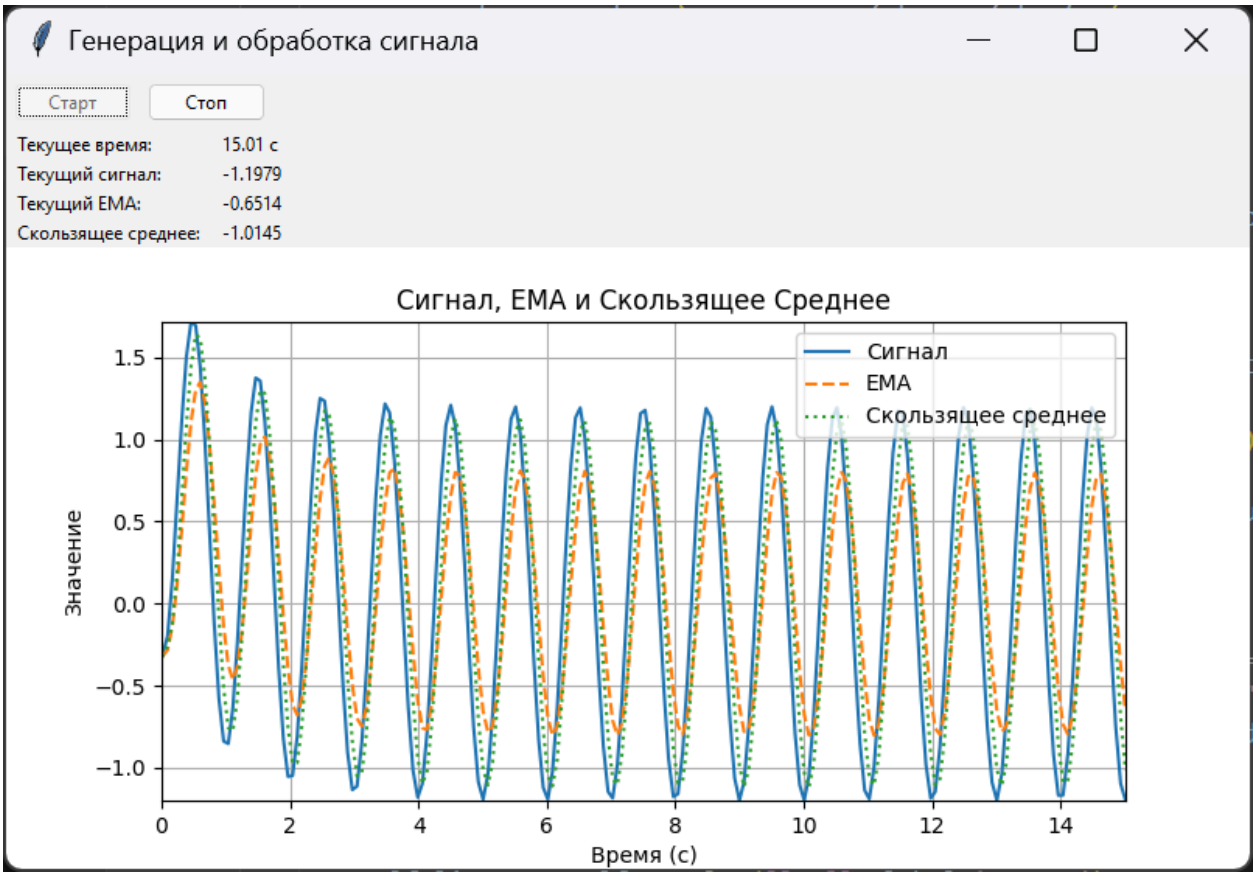

Рисунок 10 – Визуализированный сигнал после 15 секунд

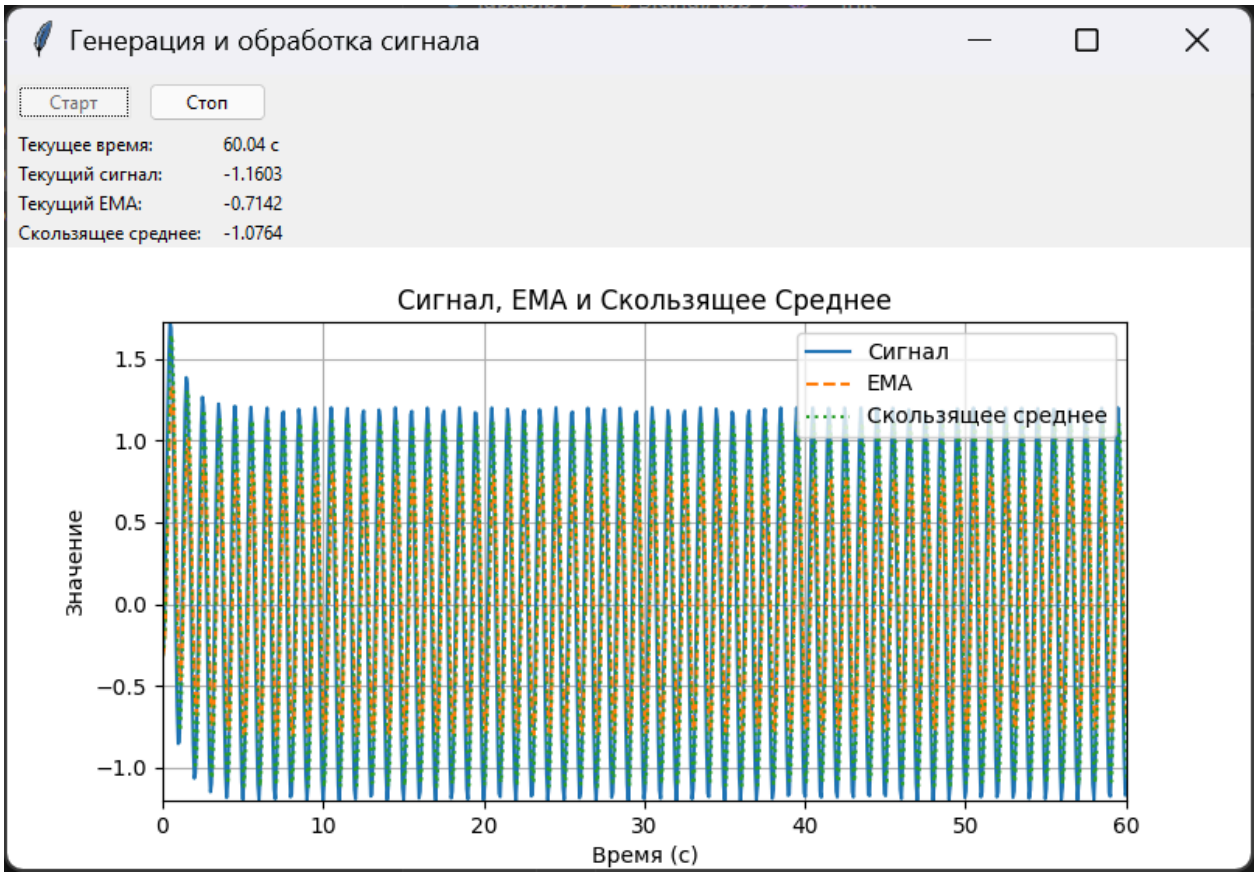

Рисунок 11 – Визуализированный сигнал после 1 минуты

Вывод: Разработана программа для генерации сложных сигналов. Она позволяет в реальном времени рассчитывать скользящее среднее и EMA, предоставляя удобный интерфейс для анализа динамических процессов.