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 №4

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

Выполнил студент 1 **курса, группы** ТИСа-о24 Блохин Э.Е.
Фамилия, имя, отчество

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

«_____» _____2024 г. _____
Подпись

Новочеркасск, 2024 г

Лабораторная работа №4

«Обеспечение защиты и хранения данных ИС и АС»

Цель работы: Разработать структуру хранилища данных, подключить инструменты хранения данных (базе данных) типа SQL или NoSQL и обеспечить безопасное хранение или передачу простым шифрованием.

Теоретический материал: Современные базы данных могут быть разделены на два основных типа: SQL и NoSQL. SQL базы данных, такие как MySQL и PostgreSQL, основаны на реляционной модели и используют язык запросов SQL для управления данными. Они обеспечивают строгую структуру данных и поддерживают транзакции, что делает их идеальными для приложений, требующих надежности и согласованности.

NoSQL базы данных, такие как MongoDB и Cassandra, предлагают более гибкую структуру данных, позволяя хранить данные в формате JSON, графов или ключ-значение. Эти системы лучше справляются с горизонтальным масштабированием и могут обрабатывать большие объемы неструктурированных данных, что делает их подходящими для приложений, где данные могут быстро изменяться.

Проектирование структуры баз данных включает в себя создание схемы, описывающей взаимосвязи между сущностями. Важно учитывать нормализацию, чтобы минимизировать дублирование данных и повысить эффективность хранения. CASE-средства помогают визуализировать и проектировать эти структуры, упрощая процесс разработки.

Защита данных в современных системах реализуется с помощью методов шифрования, которые обеспечивают безопасность данных при передаче и хранении. Алгоритмы, такие как AES и RSA, используются для шифрования данных, а также могут применяться расширения, такие как pg_crypto для PostgreSQL, для защиты информации на уровне базы данных. Протоколы передачи, такие как HTTPS, дополнительно обеспечивают защиту данных при их перемещении между клиентом и сервером.

Ход работы:

1) Разработаны две схемы: схема классов и схема последовательности, которые описывают взаимодействие компонентов системы SignalApp с базой данных. На схеме классов показано, как SignalApp использует класс Database для работы с базой данных через методы, такие как connect_to_database() для установления соединения, create_measurements_table() для создания таблицы и insert_measurement() для записи данных. Схема последовательности иллюстрирует процесс, начинающийся с вызова connect_to_database() для установления соединения с базой данных, после чего вызывается create_measurements_table() для создания таблицы хранения данных. В процессе генерации сигнала SignalApp регулярно вызывает insert_measurement() для записи значений сигнала, EMA и скользящего среднего в базу данных, а база данных подтверждает успешную вставку данных. Этот процесс обеспечивает надежное сохранение и обработку данных в реальном времени, что иллюстрируется на рисунках 1 и 2.

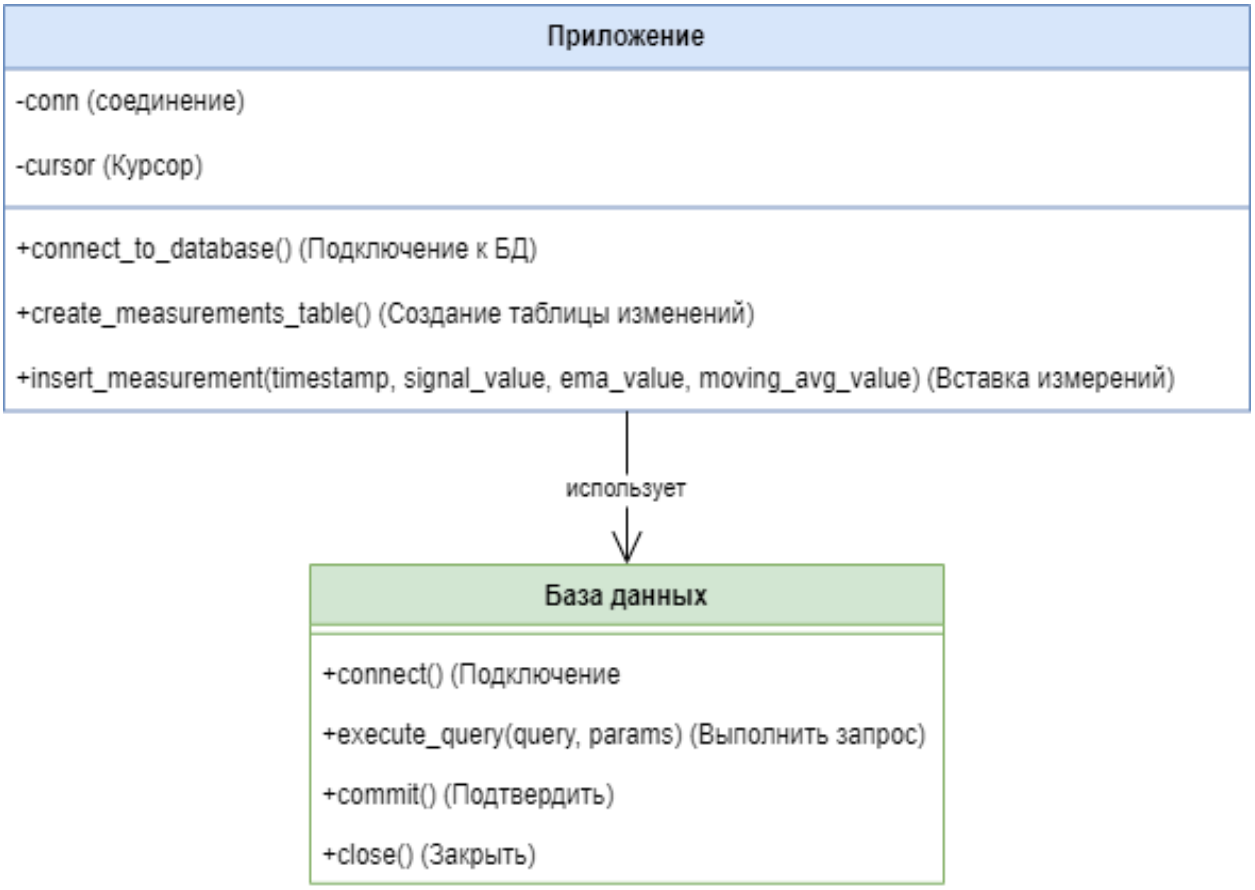
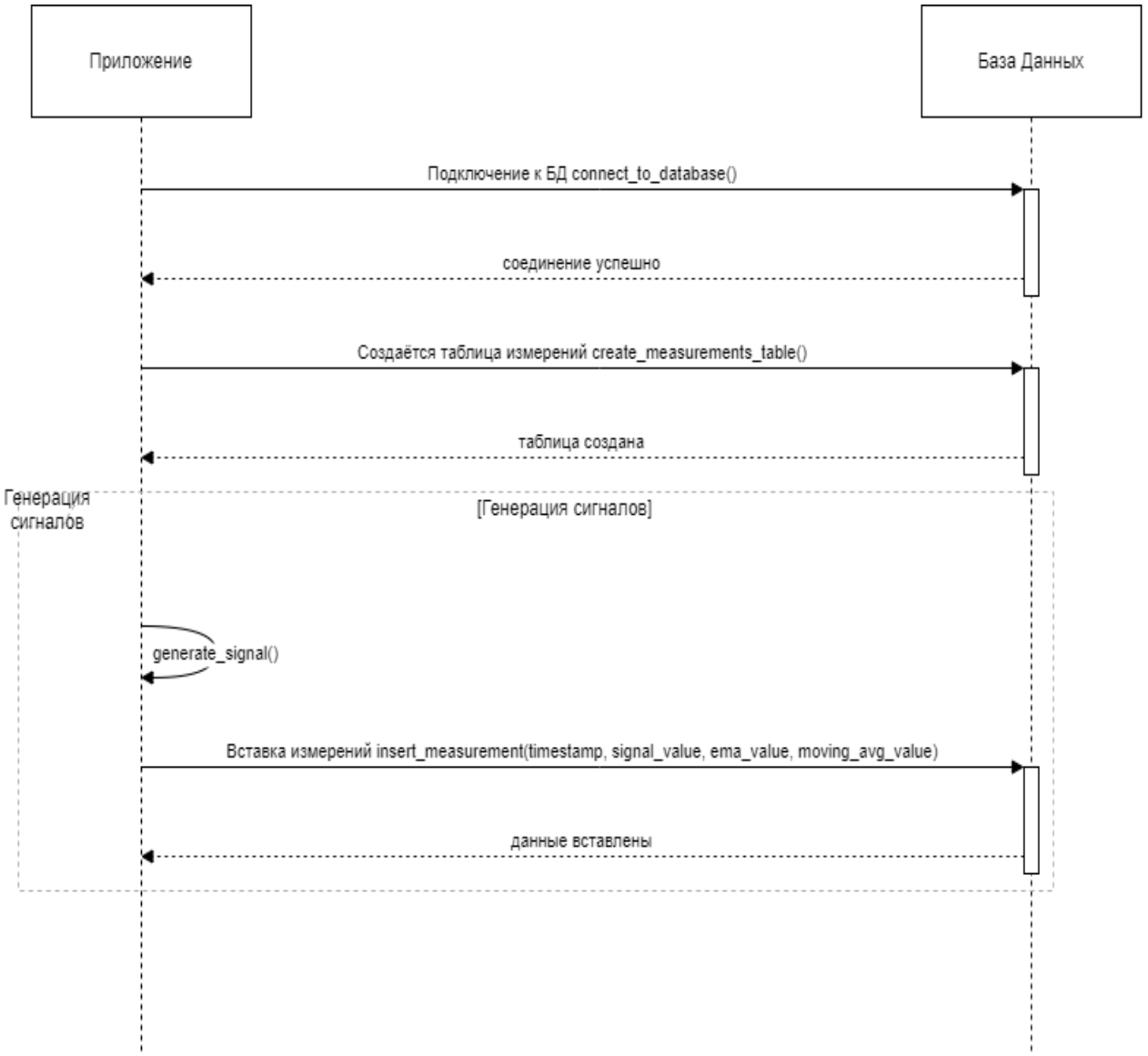


Рисунок 1 – Схема классов



массивами и функциями для обработки сигнала. Для подключения к внешним источникам данных добавлены библиотеки для работы с API (requests) и базой данных SQL Server (pyodbc). Основная функция в этой части — create_signal_single_point, которая генерирует сигнал на определённой временной точке. Она использует три разных математических компонента для формирования сигнала: экспоненциальные, косинусные и логарифмические функции. Для экспоненциальных компонентов сигнал уменьшается по мере увеличения времени, косинусные компоненты добавляют колебания, а логарифмические компоненты — медленный рост, обеспечивая сложную структуру сигнала. Эта функция сигнала представлена на рисунке 3.

```python
# Функция для генерации сигнала в один момент времени
def create_signal_single_point(time_point, num_exp, num_cos, num_log, amp_exp, amp_cos, amp_log):
    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 3 – Импорт библиотек и создание функции генерации сигнала

3) Во второй части кода создаётся класс SignalApp, который отвечает за создание графического интерфейса для генерации и обработки сигналов. В конструкторе класса инициализируются параметры сигнала, а также амплитуды для каждого из них. Задаются параметры для расчёта экспоненциального скользящего среднего (EMA) и скользящего среднего, а также инициализируются массивы для хранения значений времени, сигнала и вычисленных средних. Эта часть отображена на рисунке 4.

```python
# Класс для графического интерфейса
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 = [0.35, 0.25, 0.4]
        self.amp_cos = [1]
        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 4 – Создание класса SignalApp и инициализация параметров

4) В третьей части кода задаются параметры времени, такие как интервал дискретизации и общая продолжительность сигнала. Также инициализируются элементы графического интерфейса через метод create_widgets. Важно отметить, что здесь происходит подключение к базе данных с помощью метода connect_to_database, который устанавливает соединение с локальной базой данных SQL Server. Подключение к базе данных показано на рисунке 5.

```
# 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.create_widgets()

# Инициализация источника данных
self.data_source = "mathematical_model"  # Возможные значения: "analog_sensor", "digital_sensor", "mathematical_model", "api"

# Настройки подключения к базе данных
self.conn = None
self.cursor = None
self.connect_to_database()
```

Рисунок 5 – Инициализация интерфейса и подключение к базе данных

5) В четвёртой части кода определяется процесс подключения к базе данных и создания таблицы для хранения измерений. Метод create_measurements_table проверяет, существует ли таблица с измерениями, и если нет, создаёт её. Таблица использоваться для хранения значений сигнала, EMA и скользящего среднего во время выполнения программы (рисунок 6).

```
def connect_to_database(self):
    try:
        # Подключение к SQLite3
        self.conn = sqlite3.connect("signal_data.db")
        self.cursor = self.conn.cursor()
        print("Подключение к базе данных успешно.")
    except Exception as e:
        print(f"Ошибка подключения к базе данных: {e}")

def create_measurements_table(self):
    # Создаем таблицу, если она не существует
    create_table_query = '''
    CREATE TABLE IF NOT EXISTS Measurements (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        timestamp FLOAT,
        signal_value FLOAT,
        ema_value FLOAT,
        moving_avg_value FLOAT
    )
    '''
    self.cursor.execute(create_table_query)
    self.conn.commit()
```

Рисунок 6 – Подключение и создание таблицы в базе данных

6) В пятой части представлен метод insert_measurement, который отвечает за запись данных в базу. Этот метод принимает значения времени, сигнала, EMA и скользящего среднего, после чего выполняет SQL-запрос для вставки этих значений в таблицу БД, что показано на рисунке 7.

```
def insert_measurement(self, timestamp, signal_value, ema_value, moving_avg_value):
    # Вставляем значения в таблицу
    insert_query = '''
    INSERT INTO Measurements (timestamp, signal_value, ema_value, moving_avg_value)
    VALUES (?, ?, ?, ?)
    '''
    self.cursor.execute(insert_query, (timestamp, signal_value, ema_value, moving_avg_value))
    self.conn.commit()

def create_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)
```

Рисунок 7 – Вставка измерений в базу данных

7) В шестой части создаётся интерфейс пользователя. Интерфейс включает в себя метки для отображения текущего времени, значений сигнала, EMA и скользящего среднего. Также создаётся график для визуализации изменений данных в реальном времени с использованием библиотеки Matplotlib. Этот интерфейс показан на рисунке 8.

```
# 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 8 – Создание элементов управления и отображения значений

8) В седьмой части описаны методы start_signal и stop_signal, которые управляют запуском и остановкой процесса генерации сигнала. Метод start_signal инициализирует время, очищает массивы данных и запускает поток для генерации сигнала, тогда как stop_signal завершает этот процесс. Запуск и остановка генерации сигнала показаны на рисунке 9.

```python
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)

        # Создаем таблицу в БД
        self.create_measurements_table()

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

Рисунок 9 – Управление процессом генерации сигнала

9) В восьмой части кода метод get_signal_from_source отвечает за получение сигнала из выбранного источника. В зависимости от источника данных метод возвращает соответствующее значение сигнала. Этот процесс проиллюстрирован на рисунке 10.

```python
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 create_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://127.0.0.1:5000/signal')  # Замените на настоящий URL API
            return response.json().get('signal', 0)  # Предполагается, что API возвращает JSON с ключом 'signal'
        except Exception as e:
            print(f"Ошибка при получении данных из API: {e}")
            return 0
```

Рисунок 10 – Получение данных сигнала из различных источников

10) В девятой части кода метод generate_signal реализует непрерывную генерацию сигнала, расчёт EMA и скользящего среднего, а также обновление интерфейса в реальном времени. Внутри метода данные записываются в базу данных, и обновляется график. Важно отметить, что динамические изменения графика и текстовых меток в интерфейсе происходят во время выполнения программы. Этот процесс показан на рисунке 11.

```
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)

        # Вставка измерений в базу данных
        self.insert_measurement(self.current_time, signal_value, self.EMA_prev, 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 11 – Генерация сигнала и обновление интерфейса

11) В финальной части кода происходит настройка динамических пределов осей для графика, который обновляется в реальном времени. Метод set_xlim устанавливает горизонтальные пределы оси X, чтобы график охватывал весь временной диапазон, прошедший с начала генерации сигнала. Метод set_ylim автоматически адаптирует пределы оси Y, чтобы учесть минимальные и максимальные значения для сигнала, EMA и скользящего среднего. Это помогает избежать ситуаций, когда значения по оси Y остаются

неизменными и график выглядит неподвижным. Всё это продемонстрировано на рисунке 12.

```
        # 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}")

        # Вывод данных в терминал
        print(f"Время: {self.current_time:.2f} с, Сигнал: {signal_value:.4f}, EMA: {self.EMA_prev:.4f}, Скользящее среднее: {moving_avg:.4f}")

        time.sleep(self.sampling_interval)

def __del__(self):
    # Закрываем соединение с БД при завершении программы
    if self.cursor:
        self.cursor.close()
    if self.conn:
        self.conn.close()
```

Рисунок 12 – Генерация сигнала и обновление интерфейса

Результаты работы:

После нажатия кнопки "Старт" запускается процесс генерации сигнала, который состоит из экспоненциальных, косинусных и логарифмических компонентов, и отображается на графике в реальном времени. На графике по оси X показывается время, а по оси Y — значения сигнала, EMA и скользящего среднего, с соответствующими линиями: сплошной для сигнала, пунктирной для EMA и точечной для скользящего среднего. Одновременно обновляются текстовые метки с текущими значениями сигнала, EMA и скользящего среднего. График и текстовые поля интерфейса обновляются в реальном времени, а результаты выводятся в терминал. Кроме того, данные о времени, сигнале, EMA и скользящем среднем записываются в базу данных для дальнейшего анализа, что продемонстрировано на рисунках 13 и 14.

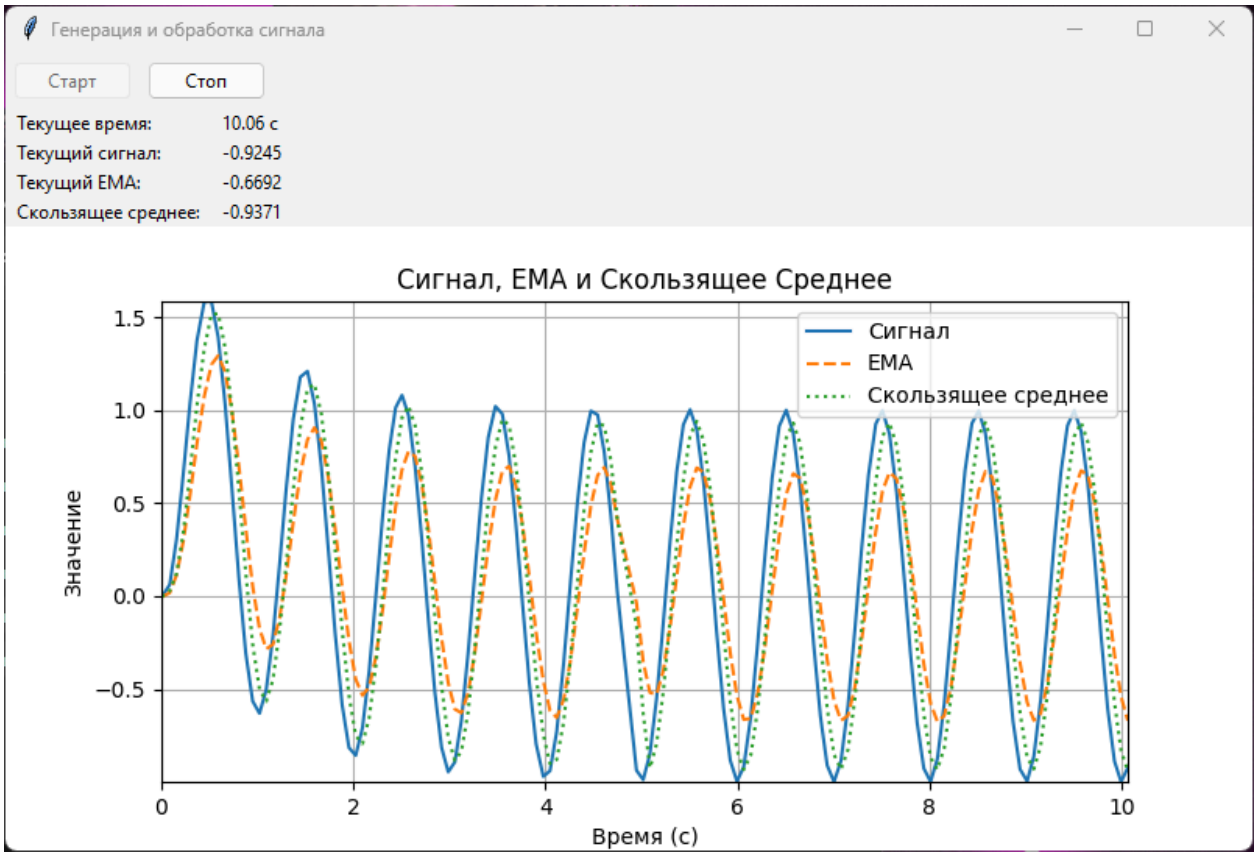


Рисунок 13 – Визуализированный сигнал и данные за 10 секунд

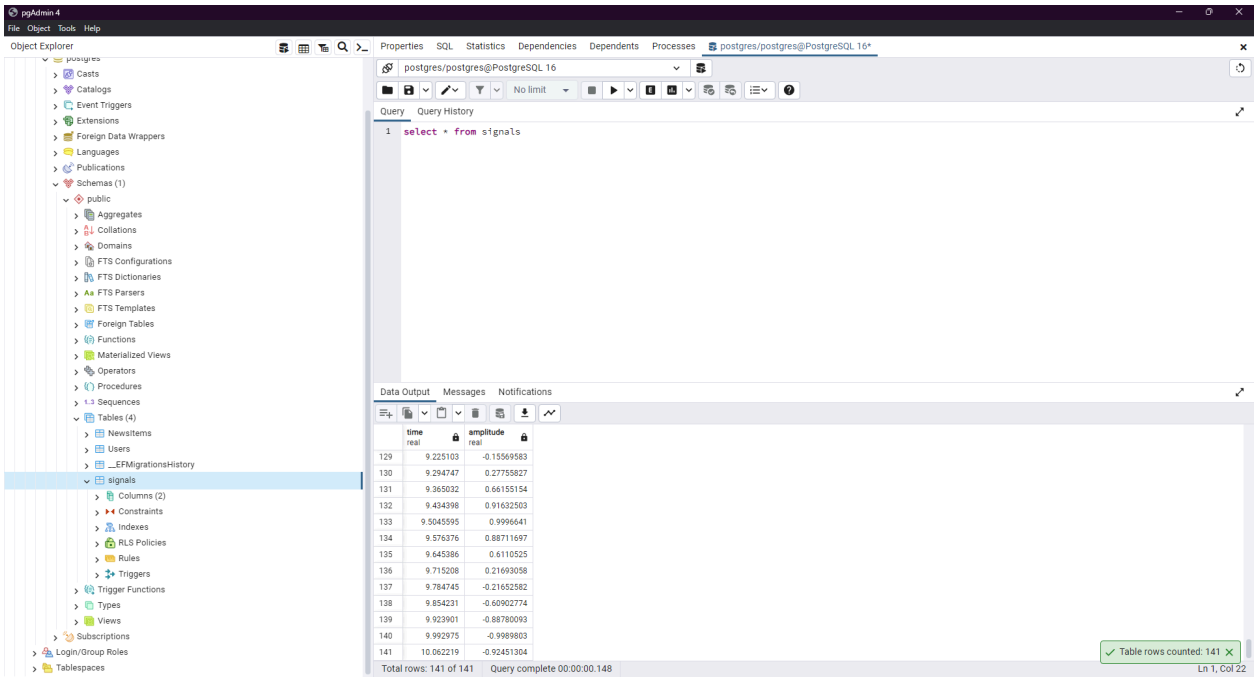


Рисунок 14 – Данные, перенесённые в БД

Вывод: Разработана программа для генерации и визуализации сложных сигналов в реальном времени с расчетом экспоненциального скользящего

среднего (EMA) и простого скользящего среднего (SMA). Программа обеспечивает удобный интерфейс для наблюдения и анализа сигналов, обновляющихся в реальном времени, с возможностью записи данных в базу для последующего анализа.