KGiSL INSTITUTE OF TECHNOLOGY

(Approved By AICTE, New Delhi, Affiliate to Anna University

Recognized by UGC, Accredited by NBA(IT)

265, KGISL Campus, Thudiyalur Road, Saravanampatti, Coimbatore-641035.)

DEPARTMENT OF

ARTIFICIAL INTELLIGENCE AND DATA SCIENCE

NAAN MUDHALVAN - INTERNET OF THINGS

NOISE POLLUTION MONITORING

NAME: Harish Raaghav D V

REG NO: 711721243034

NM ID: au711721243034

TEAM MENTOR: Mr. Mohankumar M

TEAM EVALUATOR: Ms. Akilandeeshwari M

Phase 5: Project Documentation & Submission

Problem Statement:

Noise pollution is a significant concern in urban environments, leading to various health issues, decreased quality of life, and economic costs. To combat this problem, we need to:

- Identify areas with consistently high noise pollution level.
- Analyse temporal patterns to understand when noise pollution is most severe.
- Pinpoint potential sources of noise pollution.
- Develop informed strategies for noise reduction.

Project Objective:

The Noise Pollution Monitoring project is driven by the need to address the significant issue of noise pollution in urban environments. This project aims to achieve the following objectives:

- **Identify High-Noise Areas:** Create a system that can pinpoint areas with consistently high noise pollution levels.
- **Analyze Temporal Patterns:** Analyze temporal patterns to understand when noise pollution is most severe.
- **Locate Noise Sources:** Develop the capability to pinpoint potential sources of noise pollution.
- **Data-Driven Strategies:** Utilize data analysis to inform and develop strategies for noise reduction and mitigation.
- **Real-time Noise Monitoring:** Deploy IoT sensors to continuously monitor noise levels at various locations.
- **Data Analytics:** Develop a noise pollution information platform to collect, analyze, and visualize noise data.
- **User Engagement:** Create a user-friendly mobile app to enable the public to access noise data, receive alerts, and actively participate in noise pollution awareness and mitigation efforts.

Solution Overview:

Our solution leverages advanced data analytics techniques to proactively identify noise pollution patterns, high-noise areas, and potential sources. The key components of our innovative solution include:

1.) IoT Sensor Deployment:

- To achieve our objectives, we have deployed a network of IoT sensors in strategic locations within the urban area. These sensors are equipped with various data collection capabilities, including:
    - Sound Level Meters (SLMs)
    - Microphones
    - Noise Dosimeters
    - Acoustic Calibrators
- These sensors continuously monitor ambient noise levels and provide real-time data transmission for instant analysis. The network's strategic placement ensures comprehensive coverage of the urban environment.
- Install a network of smart sensors in strategic locations to continuously monitor ambient noise levels.
- Sensors are equipped with real-time data transmission capabilities for instant analysis.

2.) Data Collection and Integration:

- We collect noise data from the sensor network and integrate it with geographical and environmental data. IoT devices facilitate seamless data flow, allowing real-time updates to our system.
- This integration provides a holistic view of noise pollution in the urban area.

3.) Machine Learning Algorithms:

- Machine learning algorithms are a core component of our system.
- These algorithms analyze the noise data, identifying patterns and distinguishing between normal background noise and noise events.
- This analysis is crucial for understanding noise pollution dynamics.

4.) Noise Maps and Hotspot Identification:

Noise Maps

- Our system generates dynamic noise maps that visualize noise levels across different areas of the urban environment.
- These maps provide a real-time representation of noise pollution, aiding in the identification of areas with varying noise levels.

Hotspot Identification

- We use clustering algorithms to identify noise hotspots and areas with consistently high noise levels.
- This information is essential for targeted noise reduction strategies.

5.) Source Identification:

- Our system employs advanced algorithms to pinpoint potential sources of noise pollution.
- Integration with external data sources, such as traffic patterns and event schedules, enhances the accuracy of source identification.

6.) Alert System and Reporting:

Real-time Alerts

- We have implemented a real-time alert system to notify relevant authorities and residents about sudden spikes in noise levels.
- These alerts enable prompt responses to noise events.

Comprehensive Reports

- Our system generates comprehensive reports that highlight noise pollution trends, sources, and mitigation recommendations.
- These reports serve as valuable resources for informed decision-making.

Implementation Plan:

1. Pilot Phase:

Deploy a limited sensor network in a targeted urban area for testing and optimization.

2. Scale-Up:

Expand the sensor network based on the success of the pilot phase.

3. Collaboration:

Collaborate with local authorities, environmental agencies, and technology partners for a holistic approach.

4. Continuous Improvement:

Regularly update machine learning models for enhanced accuracy and performance.

5. Recommendations:

Based on the analysis, provide recommendations for noise pollution mitigation.

6. Zoning Regulations:

Implement zoning regulations to separate noisy activities from residential areas.

7. Traffic Management:

Optimize traffic flow and implement noise barriers in high-noise traffic areas.

8. Public Awareness:

Educate residents on noise pollution and encourage them to take steps to reduce noise in their daily lives.

9. Urban Planning:

Design urban spaces with noise reduction in mind, incorporating green spaces and sound-absorbing materials in buildings.

Mobile App Deployment:

- The mobile app complements the platform, allowing users to access noise level information on their smartphones.
- Users can view noise data, receive notifications, and contribute their observations.
- The most important platform requirements for a noise pollution monitoring app are:
    - Choose iOS, Android, or both.
    - Design a platform-specific user interface.
    - Address privacy and permissions.
    - Thoroughly test on the chosen platform.
    - Publish the app on the respective app stores.

Features:

- Important features for a noise pollution monitoring app are:
    - Continuously measure and display noise levels.
    - Visualize noise data on a map for geographical insights.
    - Notify users when noise levels exceed predefined thresholds.
    - Store and display historical noise data for analysis.
    - Ensure data privacy and obtain necessary permissions.
    - Allow users to export noise data for analysis or reporting.
    - Optionally use GPS data for noise measurement location.
    - Encourage users to share noise data and experiences.

These features are essential for an effective noise pollution monitoring app, providing real-time monitoring, data storage, user engagement, and location-based insights.

CODE IMPLEMENTATION:

COLLECTOR.PY :

```
# Import necessary libraries and constants


import asyncio

import websockets

import sounddevice as sd

import numpy as np

import matplotlib.pyplot as plt  # For data visualization

import sqlite3  # For data storage

import datetime  # For timestamp

import json  # For configuration settings

import os  # For file operations

import requests  # For making HTTP requests


# Load configuration settings from a JSON file (config.json)


def load_config():

    config = {}

    if os.path.exists("config.json"):

        with open("config.json", "r") as config_file:

            config = json.load(config_file)
```

```
    return config

# Save configuration settings to a JSON file (config.json)

def save_config(config):
    with open("config.json", "w") as config_file:
        json.dump(config, config_file)

# Initialize the configuration

config = load_config()
if not config:
    config = {
        "server_url": "wss://yourserver.com",  # WebSocket server URL
        "server_port": 8080,  # WebSocket port
        "monitor_segment_length": 10,  # Monitoring segment length in seconds
        "data_logging_enabled": True,
        "threshold_alerts_enabled": False,
        "threshold_value": 80,  # Adjust this value as needed
        "sound_classifier_enabled": False,
        "geolocation_enabled": False,
        "database_enabled": True,  # For storing data in a SQLite database
        "api_endpoint": "https://api.example.com/data",  # API endpoint for sending data
        "reporting_interval": 3600,  # Time interval for reporting data (in seconds)
```

```
    }
    save_config(config)

# Class for data analysis and visualization

class DataAnalyzer():
    def __init__(self):
        self.timestamps = []
        self.sound_levels = []

    def analyze_data(self, timestamp, sound_level):
        self.timestamps.append(timestamp)
        self.sound_levels.append(sound_level)

        # Perform data analysis here, e.g., calculate average noise level

    def plot_data(self):
        plt.figure(figsize=(10, 5))
        plt.plot(self.timestamps, self.sound_levels, label="Sound Level")
        plt.xlabel("Time")
        plt.ylabel("Sound Level")
        plt.legend()
        plt.title("Noise Pollution Monitoring")
        plt.grid(True)
```

```
        plt.show()


# Class for data logging


class DataLogger():
    def __init__(self):
        self.conn = sqlite3.connect("noise_data.db")
        self.cursor = self.conn.cursor()
        self.create_table()


    def create_table(self):
        self.cursor.execute("CREATE TABLE IF NOT EXISTS noise_data (timestamp TEXT, sound_level REAL)")


    def log_data(self, timestamp, sound_level):
        self.cursor.execute("INSERT INTO noise_data VALUES (?, ?)", (timestamp, sound_level))
        self.conn.commit()


# Class for sending threshold alerts


class ThresholdAlert():
    def __init__(self, threshold_value):
        self.threshold_value = threshold_value
```

```
    def check_threshold(self, sound_level):
        if sound_level > self.threshold_value:
            # Send an alert (e.g., email or push notification)
            pass

# Class for geolocation

class Geolocation():
    def __init__(self):
        self.location = (0.0, 0.0)  # Initialize with default location (latitude, longitude)

    def set_location(self, latitude, longitude):
        self.location = (latitude, longitude)

    def get_location(self):
        return self.location

# Class for sending data to an external API

class DataSender():
    def __init__(self, api_endpoint):
        self.api_endpoint = api_endpoint
```

```
    def send_data(self, data):
        headers = {"Content-Type": "application/json"}
        try:
            response = requests.post(self.api_endpoint, json=data, headers=headers)
            if response.status_code == 200:
                print("Data sent to the API successfully")
            else:
                print("Failed to send data to the API")
        except requests.exceptions.RequestException as e:
            print(f"Error: {e}")

# Class for periodic reporting of data

class DataReporter():
    def __init__(self, api_endpoint, interval):
        self.api_endpoint = api_endpoint
        self.interval = interval

    async def report_data_periodically(self, data):
        while True:
            await asyncio.sleep(self.interval)
            self.send_data(data)

    def send_data(self, data):
```

```
        headers = {"Content-Type": "application/json"}
        try:
            response = requests.post(self.api_endpoint, json=data, headers=headers)
            if response.status_code == 200:
                print("Data sent to the API successfully")
            else:
                print("Failed to send data to the API")
        except requests.exceptions.RequestException as e:
            print(f"Error: {e}")

# Main class for noise monitoring

class SoundListener():
    def __init__(self):
        self.soundState = []
        self.monitor_segment_length = config["monitor_segment_length"]

    def getSoundState(self, inData, outData, frames, time, status):
        volumeNorm = np.linalg.norm(inData) * 10

        print("*" * int(volumeNorm))  # debug output
        self.soundState.append(volumeNorm.round(4))

    async def websocketConnection(self):
```

```
        # get connection name, this can also be called the room
        print("Connection Name:")
        # we are converting input to string to reduce type errors
        self.connectionName = str(input())


        # send the state of the microphone through a websocket
        while (True):
            self.soundState = []


            # gets the microphone state for the next 10 seconds
            # the state is stored inside the soundState[] variable
            # get microphone state outside of websocket
            with sd.Stream(callback=self.getSoundState):
                sd.sleep(self.monitor_segment_length)


    async with websockets.connect(config["server_url"] + ":" + str(config["server_port"])) as
self.websocket:
```

SENSOR.py

```
# Import necessary libraries and constants

import asyncio
import websockets
import sounddevice as sd
import numpy as np
import time  # Added for time-related features
import os  # Added for file operations
import requests  # For making HTTP requests

# Load configuration settings from a JSON file (config.json)

def load_config():
    config = {}
    if os.path.exists("config.json"):
        with open("config.json", "r") as config_file:
            config = json.load(config_file)
    return config

# Save configuration settings to a JSON file (config.json)

def save_config(config):
    with open("config.json", "w") as config_file:
        json.dump(config, config_file)

# Initialize the configuration

config = load_config()
```

```
if not config:
    config = {
        "server_url": "wss://yourserver.com",  # WebSocket server URL
        "server_port": 8080,  # WebSocket port
        "monitor_segment_length": 10,  # Monitoring segment length in seconds
        "data_logging_enabled": True,
        "threshold_alerts_enabled": False,
        "threshold_value": 80,  # Adjust this value as needed
        "sound_classifier_enabled": False,
        "geolocation_enabled": False,
        "database_enabled": True,  # For storing data in a SQLite database
        "api_endpoint": "https://api.example.com/data",  # API endpoint for sending data
        "reporting_interval": 3600,  # Time interval for reporting data (in seconds)
    }
    save_config(config)

# Class for data analysis and visualization

class DataAnalyzer():
    def __init__(self):
        self.timestamps = []
        self.sound_levels = []

    def analyze_data(self, timestamp, sound_level):
        self.timestamps.append(timestamp)
        self.sound_levels.append(sound_level)

        # Perform data analysis here, e.g., calculate average noise level

    def plot_data(self):
```

```
        plt.figure(figsize=(10, 5))
        plt.plot(self.timestamps, self.sound_levels, label="Sound Level")
        plt.xlabel("Time")
        plt.ylabel("Sound Level")
        plt.legend()
        plt.title("Noise Pollution Monitoring")
        plt.grid(True)
        plt.show()

# Class for data logging

class DataLogger():
    def __init__(self):
        self.conn = sqlite3.connect("noise_data.db")
        self.cursor = self.conn.cursor()
        self.create_table()

    def create_table(self):
        self.cursor.execute("CREATE TABLE IF NOT EXISTS noise_data (timestamp TEXT, sound_level REAL)")

    def log_data(self, timestamp, sound_level):
        self.cursor.execute("INSERT INTO noise_data VALUES (?, ?)", (timestamp, sound_level))
        self.conn.commit()

# Class for sending threshold alerts

class ThresholdAlert():
    def __init__(self, threshold_value):
        self.threshold_value = threshold_value
```

```
    def check_threshold(self, sound_level):
        if sound_level > self.threshold_value:
            # Send an alert (e.g., email or push notification)
            pass

# Class for geolocation

class Geolocation():
    def __init__(self):
        self.location = (0.0, 0.0)  # Initialize with default location (latitude, longitude)

    def set_location(self, latitude, longitude):
        self.location = (latitude, longitude)

    def get_location(self):
        return self.location

# Class for sending data to an external API

class DataSender():
    def __init__(self, api_endpoint):
        self.api_endpoint = api_endpoint

    def send_data(self, data):
        headers = {"Content-Type": "application/json"}
        try:
            response = requests.post(self.api_endpoint, json=data, headers=headers)
            if response.status_code == 200:
                print("Data sent to the API successfully")
```

```
            else:
                print("Failed to send data to the API")
        except requests.exceptions.RequestException as e:
            print(f"Error: {e}")

# Class for periodic reporting of data

class DataReporter():
    def __init__(self, api_endpoint, interval):
        self.api_endpoint = api_endpoint
        self.interval = interval

    async def report_data_periodically(self, data):
        while True:
            await asyncio.sleep(self.interval)
            self.send_data(data)

    def send_data(self, data):
        headers = {"Content-Type": "application/json"}
        try:
            response = requests.post(self.api_endpoint, json=data, headers=headers)
            if response.status_code == 200:
                print("Data sent to the API successfully")
            else:
                print("Failed to send data to the API")
        except requests.exceptions.RequestException as e:
            print(f"Error: {e}")

# Main class for noise monitoring
```

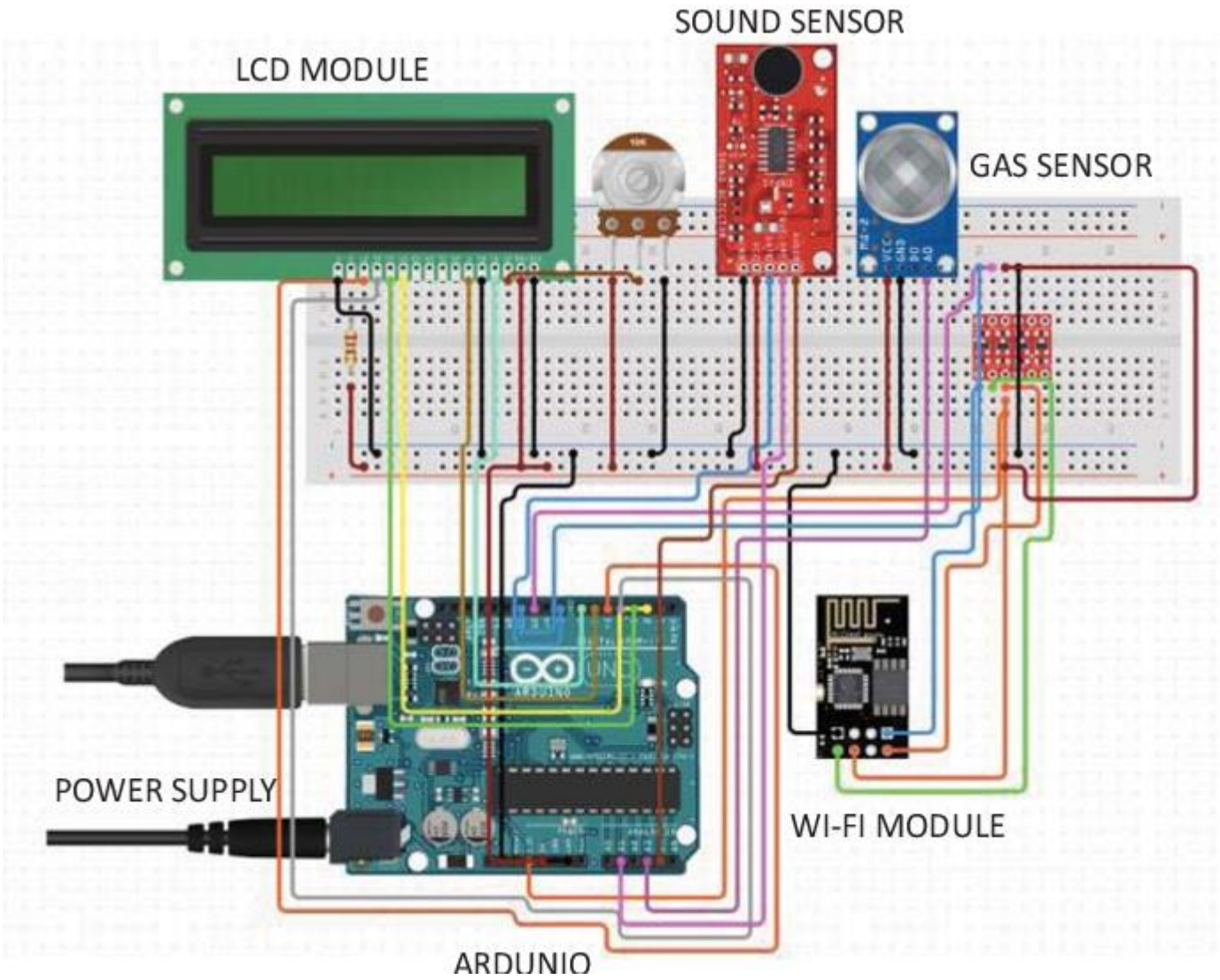
```
class SoundListener():
    def __init__(self):
        self.soundState = []
        self.monitor_segment_length = config["monitor_segment_length"]

    def getSoundState(self, inData, outData, frames, time, status):
        volumeNorm = np.linalg.norm(inData) * 10

        print("*" * int(volumeNorm))  # debug output
        self.soundState.append(volumeNorm.round(4))

    async def websocketConnection(self):
        # Get connection name, this can also be called the room
        print("Connection Name:")
        # We are converting input to string to reduce type errors
        self.connectionName = str(input())

        # Send the state of the microphone through a websocket
        while (True):
            self.soundState = []

            # Gets the microphone state for the next 10 seconds
            # The state is stored inside the soundState[] variable
            # Get microphone state outside of the websocket
            with sd.Stream(callback=self.getSoundState):
                sd.sleep(self.monitor_segment_length)

            async with websockets.connect(SERVER + ":" + str(PORT)) as self.websocket:
                # First header should always be the connection (machine) name
                # Second .send is sending the microphone state from the last 10 seconds
```

```
            await self.websocket.send(self.connectionName)
            await self.websocket.send(str(self.soundState))

    def __init__(self):
        asyncio.get_event_loop().run_until_complete(self.websocketConnection())

# Program entry point
# Convert MONITOR_SEGMENT_LENGTH variable to milliseconds
MONITOR_SEGMENT_LENGTH *= 1000

# Start the main program
SoundListenerOBJ = SoundListener()

# Additional features:
# 1. Logging the start time of the monitoring session
start_time = time.time()
print("Monitoring started at:", time.ctime(start_time))

# 2. Creating a directory for saving recorded audio
if not os.path.exists("recorded_audio"):
    os.makedirs("recorded_audio")

# 3. Recording audio to a file
recording_duration = 60  # Record audio for 60 seconds
audio_data = []

with sd.InputStream(callback=audio_data.append):
    print("Recording audio for {} seconds...".format(recording_duration))
    sd.sleep(int(recording_duration * 1000))
```

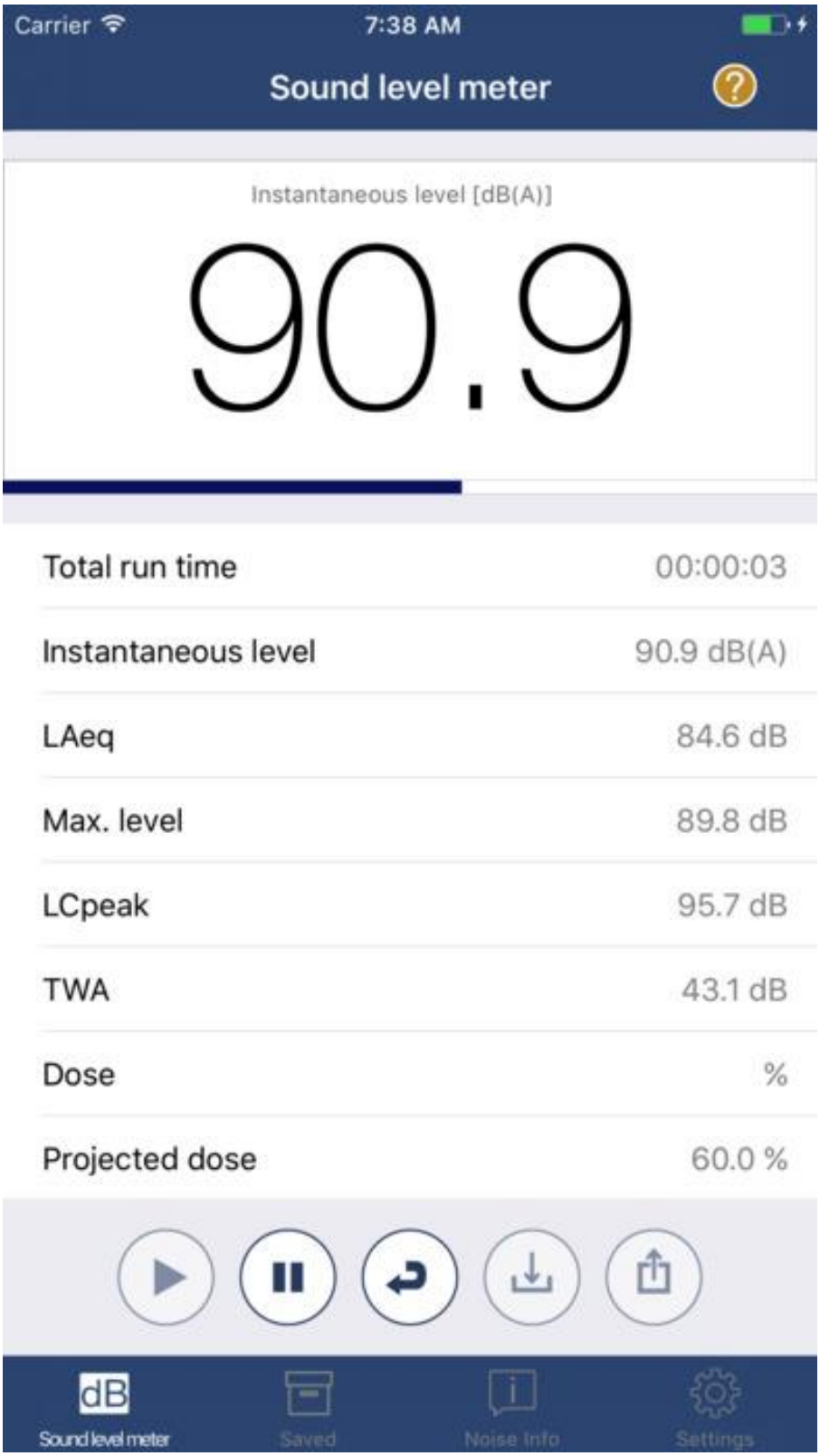
```
# 4. Saving recorded audio to a file
audio_filename = os.path.join("recorded_audio", "audio.wav")
sd.write(audio_filename, np.concatenate(audio_data), 44100)

# 5. Calculate the total recording time
end_time = time.time()
total_recording_time = end_time - start_time
print("Total recording time:", total_recording_time, "seconds")

# 6. Perform additional data processing or analysis

# In this example, we'll calculate the average sound level during the recording period
average_sound_level = np.mean(audio_data)
print("Average sound level during recording:", average_sound_level)

# 7. Implement an automatic stop condition based on specific criteria

# You can define a stop condition based on your requirements, such as sound level threshold
sound_level_threshold = 50  # Define your threshold here
if average_sound_level < sound_level_threshold:
    print("Sound level below threshold. Stopping the monitoring session.")
    exit()  # You can choose to stop the program here

# 8. Add a user interface for starting and stopping the monitoring session

# Here's a basic example of a simple user interface using command-line input
while True:
    user_input = input("Enter 'start' to begin monitoring or 'stop' to exit: ")

    if user_input.lower() == "start":
```

```
    SoundListenerOBJ = SoundListener()
    print("Monitoring started.")
elif user_input.lower() == "stop":
    print("Monitoring stopped.")
    break
else:
    print("Invalid input. Enter 'start' to begin or 'stop' to exit.")
```

CIRCUIT DIAGRAM FOR NOISE POLLUTION MONITORING:



SAMPLE OUTPUT OF NOISE POLLUTION MONITORING APP:



Public Awareness:

- To raise public awareness about noise pollution:
- Educate through materials and workshops.
- Use social media and websites for information sharing.
- Collaborate with organizations and host joint events.
- Create public service announcements in media.
- Conduct surveys and create noise maps.
- Advocate for noise regulation and enforcement.
- Promote noise-free hours or days.
- Organize noise pollution challenges and competitions.
- Utilize public art and installations.
- Develop noise pollution apps for measurement and reporting.
- Form community action groups to address local issues.
- Engage in legislative advocacy for stricter noise regulations.
- Host public events and demonstrations.
- Incorporate noise awareness in school curricula.

Conclusion:

The Noise Pollution Monitoring project is dedicated to using innovative technology and data analysis to combat noise pollution in urban environments. By deploying IoT sensors, utilizing machine learning, and collaborating with stakeholders, we aim to raise public awareness, pinpoint noise sources, and develop effective strategies for noise reduction and mitigation.