Opis działania systemu diagnostycznego lokomotywy PUTrain

Opis działania aplikacji

W opracowaniu przedstawiono fragment kodu odpowiadający za analizę danych, pozostała część kodu źródłowego powstała w ramach innego projektu, nie związanego z programem Studenckie Koła Naukowe tworzą Innowacje oraz jest własnością zespołu PUTrain i Politechniki Poznańskiej.

Aplikacja służy do **ciągłego monitorowania** drgań elementów układu napędowego lokomotywy. Przetwarza sygnał z 8 przetworników drgań, analizuje jego intensywność (RMS), określa stan techniczny każdego komponentu i przewiduje potencjalne przyszłe uszkodzenia.

Sekwencja działania krok po kroku:

1) **Start aplikacji**

- Tworzy się główne okno z 8 komponentami.
- Wczytywana jest historia RMS z pliku rms_history.json.

2) **Co 1 sekundę:**

   Dla każdego komponentu:

   - Generowany jest surowy sygnał (symulacja).
   - Filtrowany przez filtr low-pass.
   - Obliczana jest wartość RMS.
   - Porównywana z progami (limit, krytyczny).
   - Aktualizowany jest pasek i kolor w wizualizacji.

3) **Co 10 sekund:**

- Dane z sesji są zapisywane do historii.
- Historia zawęża się do ostatnich 100 pomiarów.
- Na jej podstawie aktualizowane są nowe progi RMS.

4) **Na żądanie użytkownika:**

- Można otworzyć wykresy RMS.
- Można wyeksportować historię do pliku CSV.

Aplikacja diagnostyczna została zaprojektowana z myślą o monitorowaniu i predykcji stanu technicznego integralnych podzespołów układu napędowego lokomotywy, tj. silników trakcyjnych oraz przekładni. Głównym źródłem informacji diagnostycznych są sygnały drgań pozyskiwane z akcelerometrów (czujników EVAL-ADXL1002Z), które poprzez moduł akwizycji danych NI 9205 (32-kanałowy moduł AI) trafiają do aplikacji zainstalowanej na komputerze diagnostycznym. Dzięki temu możliwe jest prowadzenie analizy danych w czasie rzeczywistym.

Aplikacja stale rejestruje dane z 8 kanałów drganiowych – 4 dla silników oraz 4 dla przekładni. Każdy sygnał jest wstępnie przetwarzany przez filtr dolnoprzepustowy (low-pass), który ogranicza analizowane pasmo do zakresu 1–200 Hz, eliminując zakłócenia

wysokoczęstotliwościowe i skupiając się na typowym paśmie diagnostycznym dla tego typu układów.

Kluczową funkcjonalnością aplikacji jest obliczanie wartości skutecznej (RMS) sygnałów drganiowych w czasie rzeczywistym. Obliczona wartość RMS dla każdego kanału porównywana jest z wartościami progowymi: dopuszczalną oraz graniczną. Te progi nie są ustalone na stałe – są one dynamicznie wyznaczane na podstawie danych historycznych i podlegają ciągłej aktualizacji wraz z pozyskaniem nowych danych pomiarowych. Dzięki temu aplikacja uczy się wzorców normalnej pracy podzespołów oraz stopniowego zużycia, dostosowując granice alarmowe do faktycznego stanu technicznego.

W przypadku przekroczenia wartości dopuszczalnej, aplikacja sygnalizuje potencjalne pogorszenie stanu technicznego danego podzespołu. Przekroczenie wartości granicznej traktowane jest jako stan alarmowy, sugerujący konieczność podjęcia działań serwisowych lub dalszej szczegółowej diagnostyki.

Dodatkowo, aplikacja wizualizuje na bieżąco trendy zmian wartości RMS dla każdego podzespołu na interaktywnych wykresach – użytkownik ma możliwość obserwacji wykresów liniowych RMS w czasie oraz progów dopuszczalnych i granicznych. Dzięki tej funkcjonalności możliwa jest łatwa identyfikacja trendów wzrostowych, które mogą świadczyć o zużyciu mechanicznym, niewyważeniu, rozluźnieniu mocowań lub innych zjawiskach diagnostycznych.

Jedną z najbardziej innowacyjnych funkcji aplikacji jest prognozowanie – na podstawie zebranej historii wartości RMS, aplikacja oblicza szacunkową liczbę dni pozostałych do przekroczenia wartości dopuszczalnej i granicznej dla każdego podzespołu. Predykcja ta jest aktualizowana po każdej nowej serii danych, co pozwala dynamicznie śledzić tempo zużycia elementów i zapobiegać awariom dzięki wcześniejszym interwencjom.

Kolejną ważną funkcją jest możliwość zapisu pełnej historii pomiarowej do pliku CSV. Zawiera on daty, wartości RMS, progi oraz wyniki predykcji, co pozwala na archiwizację danych, tworzenie raportów diagnostycznych oraz dalszą analizę offline.

W przyszłości możliwe będzie również rozwinięcie aplikacji o automatyczne generowanie alertów, uczenie maszynowe do klasyfikacji rodzajów uszkodzeń na podstawie sygnału drgań oraz integrację z systemami CMMS (Computerized Maintenance Management System).

Podsumowując, aplikacja łączy w sobie funkcje akwizycji danych, analizy sygnałów, diagnostyki opartej na wartości RMS, uczenia progów granicznych, predykcji czasu do awarii, interaktywnej wizualizacji oraz eksportu danych – stanowiąc kompleksowe narzędzie wspierające utrzymanie predykcyjne w przemyśle kolejowym.

Kod aplikacji

```
1.  import sys
2.  import numpy as np
3.  from scipy.signal import butter, lfilter
4.  from PyQt5.QtWidgets import (
5.  QApplication, QWidget, QLabel, QVBoxLayout, QGridLayout, QProgressBar, QPushButton,
    QFileDialog
6.  )
7.  from PyQt5.QtGui import QColor, QPalette
8.  from PyQt5.QtCore import QTimer
9.  import random
10. import os
11. import json
12. import matplotlib.pyplot as plt
13. import csv

14. # === Parametry filtru ===
15. SAMPLING_FREQ = 12.8  # Hz
16. CUTOFF_FREQ = 200.0  # Hz (low-pass)
17. HISTORY_FILE = "rms_history.json"

18. # === Filtr Butterwortha ===
19. def butter_lowpass_filter(data, cutoff, fs, order=5):
20. nyq = 0.5 * fs
21. normal_cutoff = cutoff / nyq
22. b, a = butter(order, normal_cutoff, btype='low', analog=False)
23. y = lfilter(b, a, data)
24. return y

25. # === Obliczanie RMS ===
26. def calculate_rms(signal):
27. return np.sqrt(np.mean(np.square(signal)))

28. # === Wczytaj historię RMS ===
29. def load_rms_history():
30. if os.path.exists(HISTORY_FILE):
31. with open(HISTORY_FILE, 'r') as f:
32. return json.load(f)
33. return {}

34. # === Zapisz historię RMS ===
35. def save_rms_history(history):
36. with open(HISTORY_FILE, 'w') as f:
37. json.dump(history, f, indent=2)

38. # === Wyznacz adaptacyjne progi RMS ===
39. def compute_thresholds(history, name):
40. values = history.get(name, [])
41. if len(values) < 5:
42. return 0.6, 0.8  # wartości domyślne
43. mean_rms = np.mean(values)
44. std_rms = np.std(values)
45. limit = mean_rms + 2 * std_rms
46. critical = mean_rms + 3 * std_rms
47. return limit, critical

48. # === Predykcja liczby dni do przekroczenia progów ===
49. def predict_days_to_threshold(history, name, threshold):
50. values = history.get(name, [])
51. if len(values) < 5:
52. return None
53. x = np.arange(len(values))
54. y = np.array(values)
```

```
    coeffs = np.polyfit(x, y, 1)  # regresja liniowa
    slope = coeffs[0]
    if slope <= 0:
    return None
    current_rms = values[-1]
    days = (threshold - current_rms) / slope
    return round(days, 1) if days > 0 else 0

    # === Eksport danych RMS do CSV ===
    def export_to_csv(history, filename):
    with open(filename, mode='w', newline='') as file:
    writer = csv.writer(file)
    writer.writerow(["Component", "RMS Values"])
    for component, values in history.items():
    writer.writerow([component] + values)

    # === Wykresy RMS ===
    def plot_rms_trends(history):
    for name, values in history.items():
    if len(values) < 5:
    continue
    limit, critical = compute_thresholds(history, name)
    days_to_limit = predict_days_to_threshold(history, name, limit)
    days_to_critical = predict_days_to_threshold(history, name, critical)

    plt.figure(figsize=(10, 4))
    plt.plot(values, label='RMS')
    plt.axhline(limit, color='orange', linestyle='--', label=f'Limit ({days_to_limit} dni)')
    plt.axhline(critical, color='red', linestyle='--', label=f'Krytyczny ({days_to_critical}
    dni)')
    plt.title(f'RMS Trend - {name}')
    plt.xlabel('Pomiar')
    plt.ylabel('RMS')
    plt.legend()
    plt.grid(True)
    plt.tight_layout()
    plt.show()

    # === Widget komponentu (silnik/przekładnia) ===
    class ComponentWidget(QWidget):
    def __init__(self, name):
    super().__init__()
    self.name = name
    layout = QVBoxLayout()

    self.label = QLabel(f"{name} RMS")
    self.rms_display = QProgressBar()
    self.rms_display.setRange(0, 100)

    self.status_label = QLabel("Condition state")
    self.status_light = QLabel()
    self.status_light.setAutoFillBackground(True)

    layout.addWidget(self.label)
    layout.addWidget(self.rms_display)
    layout.addWidget(self.status_label)
    layout.addWidget(self.status_light)

    self.setLayout(layout)

    def update_status(self, rms, limit, critical):
    self.rms_display.setValue(int(rms * 10))

    pal = self.status_light.palette()
    if rms < limit:
    pal.setColor(QPalette.Window, QColor('green'))
    elif rms < critical:
    pal.setColor(QPalette.Window, QColor('orange'))
```

```
112.    else:
113.    pal.setColor(QPalette.Window, QColor('red'))
114.    self.status_light.setPalette(pal)

115.    # === Główne okno aplikacji ===
116.    class DiagnosticApp(QWidget):
117.    def __init__(self):
118.    super().__init__()
119.    self.setWindowTitle("Diagnostic Dashboard")
120.    self.resize(1000, 600)

121.    layout = QGridLayout()

122.    self.components = {}
123.    self.history = load_rms_history()
124.    self.session_data = {k: [] for k in [f"ENG{i+1}" for i in range(4)] + [f"PG{i+1}" for i
        in range(4)]}

125.    names = list(self.session_data.keys())
126.    for idx, name in enumerate(names):
127.    comp = ComponentWidget(name)
128.    layout.addWidget(comp, idx // 2, idx % 2)
129.    self.components[name] = comp

130.    # Przyciski funkcjonalne
131.    self.plot_button = QPushButton("Pokaż wykresy RMS")
132.    self.plot_button.clicked.connect(lambda: plot_rms_trends(self.history))
133.    layout.addWidget(self.plot_button, 5, 0, 1, 2)

134.    self.export_button = QPushButton("Eksportuj historię do CSV")
135.    self.export_button.clicked.connect(self.export_history)
136.    layout.addWidget(self.export_button, 6, 0, 1, 2)

137.    self.setLayout(layout)

138.    self.timer = QTimer()
139.    self.timer.timeout.connect(self.update_readings)
140.    self.timer.start(1000)  # Aktualizacja co sekundę

141.    self.reading_counter = 0
142.    self.max_readings = 10  # Po ilu odczytach aktualizować historię i progi

143.    def update_readings(self):
144.    for name, widget in self.components.items():
145.    raw_signal = np.array([random.uniform(0, 1.2) for _ in range(8)])
146.    filtered = butter_lowpass_filter(raw_signal, CUTOFF_FREQ, SAMPLING_FREQ)
147.    rms = calculate_rms(filtered)

148.    # Zapisz do danych sesji
149.    self.session_data[name].append(rms)

150.    # Wyznacz progi adaptacyjne
151.    limit, critical = compute_thresholds(self.history, name)
152.    widget.update_status(rms, limit, critical)

153.    self.reading_counter += 1

154.    if self.reading_counter >= self.max_readings:
155.    self.update_history()
156.    self.reading_counter = 0

157.    def update_history(self):
158.    for name, values in self.session_data.items():
159.    if name not in self.history:
160.    self.history[name] = []
161.    self.history[name].extend(values)
162.    self.history[name] = self.history[name][-100:]
163.    self.session_data[name] = []
```

```
164.    save_rms_history(self.history)

165.    def export_history(self):
166.    filename, _ = QFileDialog.getSaveFileName(self, "Zapisz CSV", "rms_history.csv", "CSV
        files (*.csv)")
167.    if filename:
168.    export_to_csv(self.history, filename)

169.    if __name__ == '__main__':
170.    app = QApplication(sys.argv)
171.    window = DiagnosticApp()
172.    window.show()
173.    sys.exit(app.exec_())
```

Opis działania fragmentów kodu aplikacji

Fragment kodu	Opis działania
`import numpy as np`	Importuje bibliotekę NumPy do obliczeń matematycznych
`from scipy.signal import butter, lfilter`	Importuje funkcje do tworzenia i stosowania filtru Butterwortha
`def butter_lowpass_filter(data, cutoff, fs, order=5):`	Definiuje funkcję filtra dolnoprzepustowego
`    nyq = 0.5 * fs`	Oblicza częstotliwość Nyquista (połowa częstotliwości próbkowania)
`    normal_cutoff = cutoff / nyq`	Normalizuje częstotliwość odcięcia
`    b, a = butter(order, normal_cutoff, btype='low')`	Tworzy współczynniki filtru Butterwortha
`    return lfilter(b, a, data)`	Zwraca przefiltrowany sygnał
`def calculate_rms(signal):`	Definiuje funkcję obliczania wartości skutecznej RMS
`    return np.sqrt(np.mean(np.square(signal)))`	Oblicza pierwiastek z wartości średniej kwadratu sygnału
`def compute_thresholds(history, name):`	Wyznacza progi dopuszczalne i graniczne na podstawie historii
`    mean = np.mean(values)`	Oblicza średnią wartość RMS
`    std = np.std(values)`	Oblicza odchylenie standardowe
`    return mean + 2*std, mean + 3*std`	Zwraca odpowiednio limit dopuszczalny i krytyczny