Client industriel – GCK Battery	Département Génie Électrique
Représenté par Arnaud VOYER - Ingénieur systèmes embarqués	**Tuteur GE** Jacques LAFFONT **Tuteur industriel** Julian LAURENCE

Conception d'un banc de test de batteries

Note d'application

par

Jalil LAHRACH
Rayan REZKI

Département Génie Électrique
POLYTECH Clermont

Table des matières

Intro :

Ce document se veut être une note d'application d'une partie de ce qui a été réalisé lors du projet de conception d'un banc de test de batterie pour la branche GCK batterie l'entreprise GCK sous la supervision des enseignants de l'école Polytech'Clermont. Vous verrez comment implémenter et comprendre le programme de communication Ethernet avec l'alimentation bidirectionnelle chargée de réaliser les cycles de charge/décharge ainsi que celui de l'interface utilisateur (IHM) qui permet de faire le lien entre l'appareil et l'humain qui souhaite s'en servir.

1. Communication avec l'alimentation bidirectionnelle :

1.1. Communication Ethernet avec l'alimentation :

Comme stipulé dans le cahier des charges, l'alimentation qui sera utilisée tout au long de ce processus est de la gamme Elektro-Automatik EA-PSB 10000.

Figure 1: Image de la datasheet de l'alimentation et le logo de la marque

Le mode de communication que l'on utilisera sera un mode Ethernet. Pour cela, il est nécessaire de se connecter à l'alimentation en deux phases :

- De manière filaire, donc à l'aide d'un câble RJ45 branché au port LAN (Ethernet) de l'alimentation.

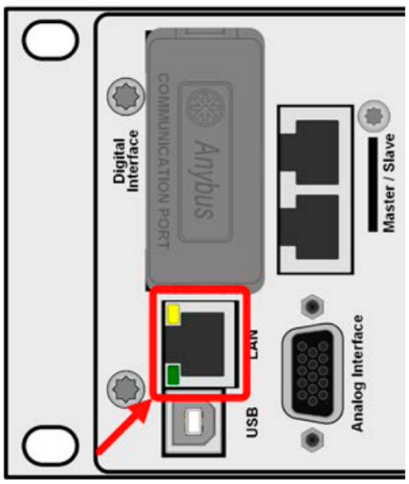

Figure 8 - LAN port

Figure 2 : Port LAN de l'alimentation bidirectionnelle EA-PSB 10000

- De manière numérique, aux mêmes réseaux que l'alimentation, donc en configurant sur l'ordinateur qui communique : le numéro du port TCP (fourni par l'alimentation), le type de configuration IP (ici, « Manuellement », pour pouvoir modifier le reste des paramètres), l'adresse IP (propre à l'ordinateur), le masque de sous-réseau et le routeur (qui doivent être les mêmes que ceux de l'alimentation).

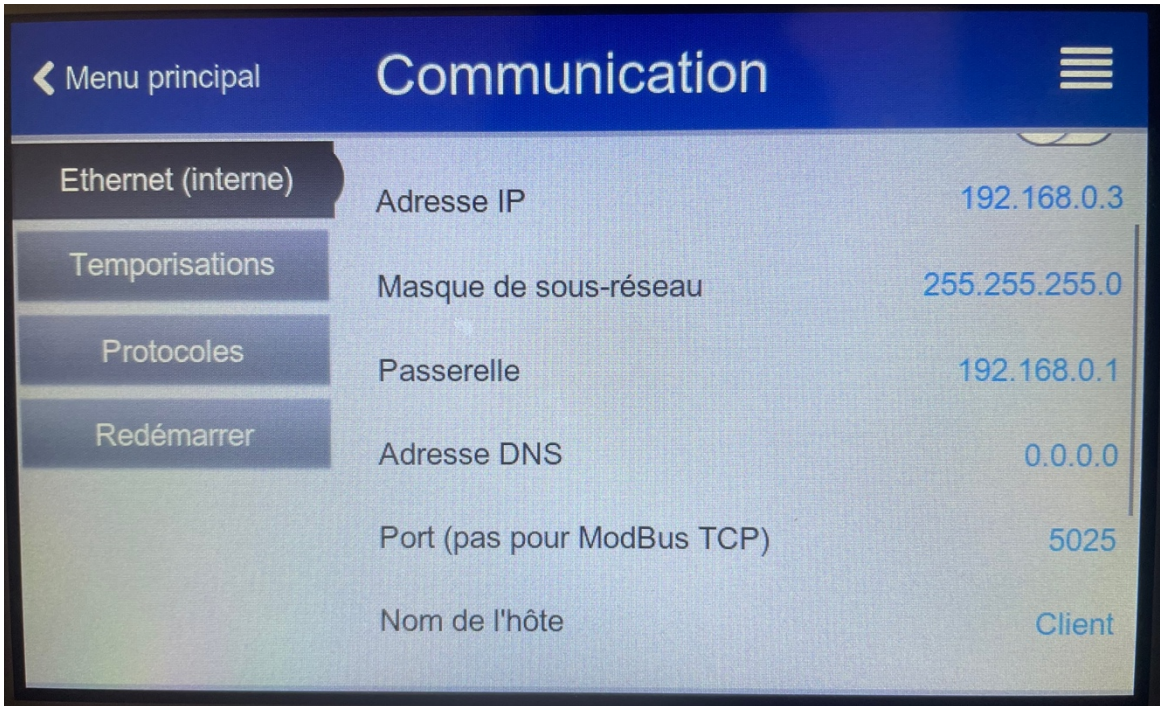

Figure 3 : Photo du paramétrage de la communication Ethernet de l'alimentation

D'après l'écran d'affichage de l'alimentation, l'adresse IP utilisée par défaut par celle-ci est « 192.168.0.3 », le routeur est « 192.168.0.1 », le numéro de port est « 5025 » et le masque de sous-réseau est « 255.255.255.0 ». Il est alors nécessaire de choisir une adresse IP pour notre ordinateur de la forme « 192.168.0.X » car le masque indique que l'identifiant de l'appareil (ce qui le différencie des autres appareils sur ce réseau) est la

dernière valeur de son adresse IP (seul le dernier numéro du masque est différent de 255). Ce choix permettra à l'alimentation de reconnaitre l'ordinateur et inversement.

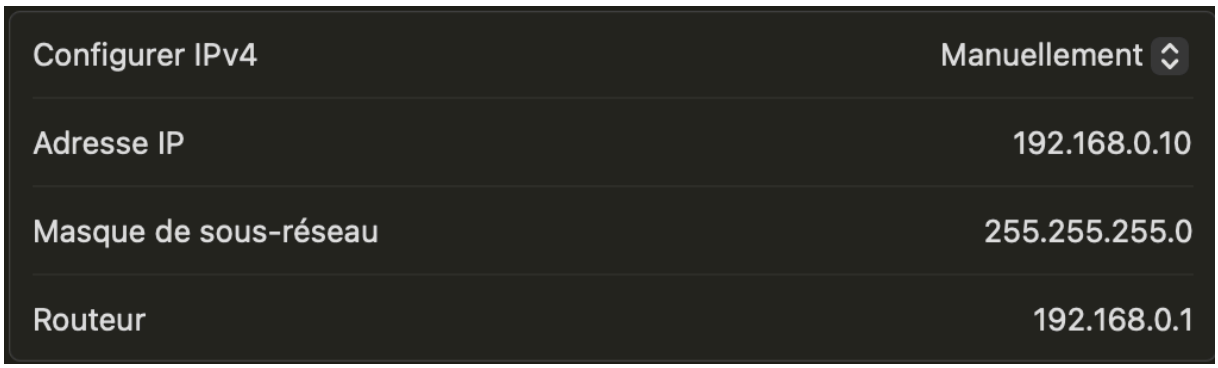

Figure 3 : Exemple de paramétrage réseaux de l'ordinateur fonctionnel

1.2. Contrôle et cycles de charge/décharge d'une alimentation via SCPI :

Dans le cadre de la programmation de l'alimentation, il est nécessaire de télécharger la bibliothèque « socket » pour utiliser la syntaxe SCPI et envoyer les commandes. De plus, la bibliothèque « time » est requise pour insérer des délais entre les commandes, permettant ainsi de gérer les cycles de charge et de décharge.

```
import socket
import time
```

Figure 4 : Import des bibliothèques utiles

1.2.1. Objectif :

Ce programme vise à établir une connexion avec une alimentation programmable via le protocole SCPI (Standard Commands for Programmable Instruments). Il permet d'effectuer des cycles de charge et de décharge en utilisant une interface en Python. La note fournit une analyse approfondie des fonctions, de la logique des cycles, et des étapes d'initialisation.

1.2.2. Structure du programme :

1.2.2.1. Paramètres globaux :

```python
# Paramètres de connexion
IP_ADDRESS = "192.168.0.3"
PORT = 5025
NUM_CYCLES = 5  # Nombre de cycles charge/décharge
```

Figure 5 : Paramètres globaux

- « IP_ADDRESS » et « PORT » : Spécifient l'adresse IP et le port TCP de l'alimentation.
- « NUM_CYCLES » : Nombre de cycles charge/décharge à exécuter.

1.2.2.2. Fonctions utilitaires :

- **Envoi de commandes :**

```python
# Fonctions utilitaires
def send_command(sock, command):
    """Envoie une commande SCPI via une connexion existante."""
    try:
        if sock.fileno() == -1:  # Vérifie si le socket est actif
            print("Le socket est fermé. Reconnexion...")
            sock = reconnect(sock)

        command = command.strip() + "\n"
        sock.sendall(command.encode())
        time.sleep(1)  # Augmentez le délai à 1 seconde
        response = sock.recv(1024).decode().strip()
        print(f"Commande: {command.strip()}\nRéponse: {response}")
        return response
    except BrokenPipeError:
        print("Connexion rompue. Tentative de reconnexion...")
        raise
    except Exception as e:
        print(f"Erreur lors de l'envoi de la commande {command.strip()}: {e}")
        raise
```

Figure 6 : Fonction d'envoie des commandes

- Cette fonction transmet une commande SCPI au dispositif via une connexion existante.
- Gestion d'exceptions :
    - Si le socket est fermé, une reconnexion automatique est tentée.
    - Les erreurs courantes (e.g., BrokenPipeError) sont capturées pour assurer la robustesse.
- Réponse : La fonction retourne la réponse du dispositif à la commande envoyée.

- **Connexion initiale :**

```python
def connect_to_device():
    """Établit une connexion au dispositif."""
    try:
        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sock.settimeout(10)
        sock.setsockopt(socket.SOL_SOCKET, socket.SO_KEEPALIVE, 1)  # Active TCP Keep-Alive
        sock.connect((IP_ADDRESS, PORT))
        print("Connecté à l'alimentation.")
        return sock
    except Exception as e:
        print(f"Erreur lors de la connexion : {e}")
        raise
```

Figure 7 : Fonction de connexion initiale

- Établit une connexion TCP avec l'alimentation en utilisant les paramètres IP et port.
- Active l'option TCP « Keep-Alive » pour maintenir la connexion active.
- « Timeout » : La fonction inclut un délai de 10 secondes pour gérer les pannes de connexion.

- **Rétablissement de connexion :**

```python
def reconnect(sock):
    """Rétablit une connexion en cas de perte."""
    try:
        sock.close()
    except:
        pass
    return connect_to_device()
```

Figure 8 : Fonction de reconnexion

- En cas de perte de connexion, cette fonction ferme le socket existant et tente une reconnexion automatique.

- **Lecture des journaux d'erreur :**

ModBus & SCPI

5.4.5.4 SYSTem:ERRor?

This commands is used to read a single error or all errors from the device's internal SCPI error queue. This queue only contains communication errors, such as about wrong syntax, excess values etc. It can not return any device alarm (one exception: SOVP). Also see section *„5.2.5. Communication and other errors“*. Device alarms can be queried from the device by reading bits of the status registers (see *„5.4.2. Status registers“*).

You can either chose to query the next error multiple times until it says "No error" or query all at once. After all errors have been read from the buffer, it will be purged.

The queue is of type FIFO (first in, first out). It means, that the first occurred error is returned first.

Query form 1:	SYSTem:ERRor?	Queries the next error
Query form 2:	SYSTem:ERRor:NEXT?	Queries the next error
Query form 3:	SYSTem:ERRor:ALL?	Queries all errors in the buffer (up to 5)

Example:

SYST:ERR?	Absolute short form. The device replies to this query with a string that first contains an error code (see error code list) and second an error description, for example: 0,"No error". This is returned every time no error is present or after all error have been returned.

Figure 9 : Commande de récupération des erreurs d'après le « Guide de Programmation » de l'alimentation

```python
def log_device_error(sock):
    """Récupère les erreurs de l'appareil."""
    try:
        error_log = send_command(sock, "SYST:ERR?")
        print(f"Log de l'appareil : {error_log}")
    except Exception as e:
        print(f"Impossible de lire le log de l'appareil : {e}")
```

Figure 10 : Fonction de diagnostic des erreurs

- Utilise la commande SCPI « SYST:ERR? » pour récupérer les logs d'erreur du dispositif.

Error code / error text	Description
0,"No error"	No error
-100,"Command error"	Command unknown
-102,"Syntax error"	Command syntax wrong
-108,"Parameter not allowed"	A command was sent with a parameter though the command doesn't use parameters
-200,"Execution error"	Command could not be executed
-201,"Invalid while in local"	Control command could not be executed, because device is in LOCAL mode
-220,"Parameter error"	Wrong parameter used
-221,"Settings conflict"	Command could not be executed because of the condition of the device being in the setup menu or isn't in remote mode yet
-222,"Data out of range"	Parameter could not be set because it exceeded a limit
-223,"Too much data"	Too many parameters per command or too many commands at once
-224,"Illegal parameter value"	A parameter not specified for the command has been sent
-225,"Out of memory"	A concatenated query has been sent whose concatenated response would exceed 256 characters (max. SCPI buffer length), such as 5x *IDN?
-999,"Safety OVP"	Exception (device alarm), because no communication error: Safety OVP (only available with 60 V models of select series) has been triggered (see device manual). It requires to power cycle the device.

Figure 11 : Décodage des erreurs d'après le "Guide de Programmation"

- **Initialisation :**

5.4.1 Standard IEEE commands

In relation to the old interface standards GPIB and IEEE 488, some of the standard commands have been implemented. They are supported in all devices which feature SCPI command language.

5.4.1.1 *CLS

Clears the error queue, the status byte (STB) and all bits in the Event Status Register (ESR), except for bit 0.

5.4.1.2 *IDN?

Returns the device identification string, which contains following information, separated by commas:

1. Manufacturer
2. Model name
3. Serial number
4. Firmware version(s) (in case there are several, these are separated by a space)
5. User text (arbitrary user-definable text, as definable with SYST:CONFIG:USER:TEXT)

5.4.1.3 *RST

When sent, this will set the device to a defined state, except remote control is denied by the device:

1. Switch to remote control (same as SYST:LOCK 1)
2. Set DC input/output to off
3. Clear alarm buffer
4. Clear status registers to default condition (QUEStionable Event, OPERation Event, STB)

Figure 12 : Description des commandes et requêtes standards IEEE

► **How to switch a device to remote control:**

1. If you are using SCPI command language, send a text command (the space is required):
   SYST:LOCK␣1 or SYST:LOCK␣ON

Leaving remote control can be done in two ways: using the dedicated command or by switching the device to "**Local**" condition. We will consider the first option, because this is about programming.

Figure 13 : Passage en mode "remote control"

```python
def initialize_device(sock):
    """Initialise le dispositif et active le contrôle à distance."""
    # Identification de l'appareil
    response = send_command(sock, "*IDN?")
    if not response:
        raise Exception("Aucune réponse de l'appareil. Vérifiez la connexion.")

    # Activation du contrôle à distance et configuration
    send_command(sock, "SYST:LOCK ON")
    send_command(sock, "SYST:TIMEOUT 0")  # Désactivation du timeout SCPI
    send_command(sock, "*CLS")  # Efface les erreurs précédentes
    send_command(sock, "*RST")  # Réinitialisation de l'appareil
```

Figure 14 : Fonction d'initialisation de l'appareil

- Identification : Utilise la requête « *IDN? » pour vérifier que l'appareil est connecté et répond correctement.

```
Commande: *IDN?
Réponse: EA Elektro-Automatik GmbH & Co. KG, ELR 10200-50, 2775120003, V3.03 05.10.2022 V3.06 24.05.2023 V1.0.2.20,
```

Figure 15 : Validation de la connexion avec l'appareil

- Configuration :
  - Active le contrôle à distance (« SYST:LOCK ON »).
  - Désactive les timeouts SCPI (« SYST:TIMEOUT 0 »).
  - Efface les erreurs précédentes avec « *CLS ».
  - Réinitialise l'appareil à son état par défaut avec « *RST ».

- **Logique des cycles :**

5.4.3.1 [SOURce:]VOLTage␣<NR2>

Sets the input or output voltage limit of the device within a certain range, which is either defined by adjustment limits ("Limits", where featured) or is 0...102% nominal value, or reads the last setting. Alternatively, parameters MIN or MAX can be used to instantly set the voltage to the adjustable MINimum or MAXimum.

Query form:	[SOURce:]VOLTage?
Value range:	<NRf> = 0...1.02 * nominal voltage (according to technical specs)
Examples:	
VOLT 12	Absolute short form. Sets 12 V.

Figure 16 : Commande de "preset" de tension pour effectuer un premier test

5.4.5.3 OUTPut_<B0>

ELR9	ELR5	PS9	PSI9	PSI5	PSE	DT	PST	PSIT	EL3	PSB	PSBE	PS3	PS2	PS1	PSI1	PSB1	PSBE1	ELR1
–	–	✓	✓	✓	✓	✓[2]	✓	✓	–	✓	✓	✓	✓	–	✓	✓	✓	–

This command is used to switch the DC output on or off with devices that have an output, like power supplies or bidirectional devices which are also considered as power supplies.

Query form: OUTPut?

Value range: ON, OFF

Examples:

OUTP_ON Absolute short form. Switches the DC output on if remote control is active.

Figure 17 : Commande d'activation et de désactivation de la sortie de l'alimentation

```python
def cycle_charge_discharge(sock):
    """Effectue les cycles de charge et décharge."""
    for cycle in range(1, NUM_CYCLES + 1):
        print(f"\nDébut du cycle {cycle}/{NUM_CYCLES}")

        # Étape 1 : Charge
        print("Démarrage de la charge...")
        try:
            time.sleep(0.5)
            send_command(sock, "VOLT 10")
            time.sleep(10)  # Temps de charge

        except BrokenPipeError:
            print("Reconnexion nécessaire pendant la charge.")
            sock = reconnect(sock)
            log_device_error(sock)
            continue

    print("\nTous les cycles sont terminés.")
    send_command(sock, "OUTP:STAT OFF")  # Désactive la sortie
```

Figure 18 : Logique de commande de l'alimentation

- Cette fonction exécute plusieurs cycles de charge et décharge selon le paramètre « NUM_CYCLES ».
- Cycle typique :

1. Charge : Configure la tension à 10V avec la commande « VOLT 10 » et attend un délai (10 secondes par défaut).
2. Gestion des erreurs :
   - En cas de perte de connexion pendant un cycle, la fonction tente une reconnexion et consigne les erreurs.
3. Désactivation de la sortie : Après tous les cycles, la sortie de l'alimentation est désactivée via « OUTP:STAT OFF ».

- **Script principal**

```python
# Script principal
try:
    sock = connect_to_device()
    initialize_device(sock)
    send_command(sock, "OUTP ON")
    cycle_charge_discharge(sock)
except BrokenPipeError:
    print("Erreur critique : Connexion rompue. Tentative de réinitialisation...")
    sock = reconnect(sock)
except Exception as e:
    print(f"Erreur critique : {e}")
finally:
    if sock:
        try:
            send_command(sock, "OUTP OFF")
            send_command(sock, "OUTP:STAT OFF")  # Assurez-vous que l'alimentation est arrêtée

        except:
            pass
        sock.close()
        print("Connexion fermée.")
```

Figure 19 : Script principal d'exécution du code

- Connexion et initialisation :

1. Une connexion est établie avec l'appareil via « connect_to_device() ».
2. L'appareil est initialisé avec « initialize_device(sock) ».
3. La sortie de l'alimentation est activée via « OUTP ON ».

- Exécution des cycles :

- Les cycles charge/décharge sont effectués avec « cycle_charge_discharge(sock) ».

- Gestion des erreurs critiques :

- En cas de perte de connexion majeure (BrokenPipeError), une tentative de reconnexion est effectuée.

- Nettoyage final :

- La sortie de l'alimentation est désactivée (à l'aide de « OUTP OFF » et « OUTP:STAT OFF »).

- Le socket est fermé proprement.

1.2.3. Ouverture :

1. Essayer le programme sur l'alimentation du client.
2. Utiliser les commandes consacrées aux batteries du « Programming Guide »

3. Lire avec attention la datasheet Hardware de l'alimentation afin de définir un algorithme pour effectuer les cycles. (Puis modifier la fonction « cycle_charge_discharge(sock) »)

Conclusion : Ce programme constitue une base solide pour l'interaction avec une alimentation programmable via SCPI. Il peut être adapté et étendu à d'autres besoins en ajoutant des commandes SCPI ou en modifiant la logique des cycles.

2. L'Interface Homme-Machine (IHM)

2.1. Description générale

Le script fournit une interface graphique (IHM) pour gérer un banc de cyclage de batteries. Il utilise la bibliothèque « tkinter » pour l'IHM, « csv » pour manipuler des fichiers de profil, et « matplotlib » pour visualiser les données collectées.

```
import tkinter as tk
from tkinter import filedialog, messagebox
import csv
import time
from threading import Thread
import matplotlib.pyplot as plt
```

Figure 20 : Bibliothèque à importer pour l'exécution de cette IHM

2.2. Fonctionnalités principales

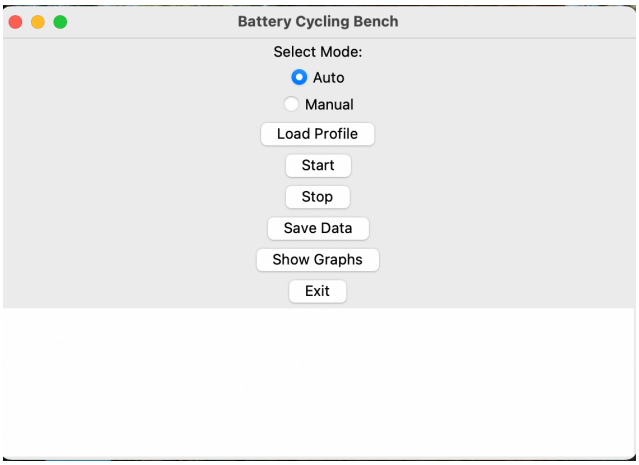

Figure 21 : Affichage de l'IHM

1. Modes d'opération

- Mode automatique : acquisition continue de données sur une durée définie. (Que des 0 en l'absence de valeurs capteur)
- Mode manuel : suivi d'un profil de cycle de charge/décharge prédéfini.

2. Gestion des profils

- Chargement d'un profil de cycle à partir d'un fichier CSV contenant des paramètres (voltage, courant, etc.).

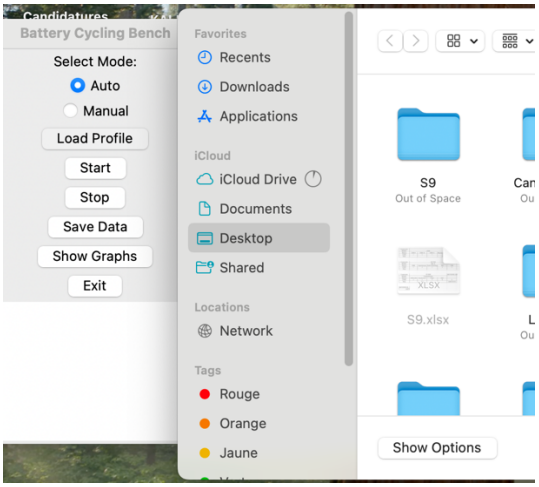

Figure 22 : Ouverture de la recherche de fichier à la suite de l'appui sur le bouton "Load Profile"

3. Acquisition de données

- Simule la capture de paramètres de la batterie (tension, courant, température, etc.) à intervalles réguliers.

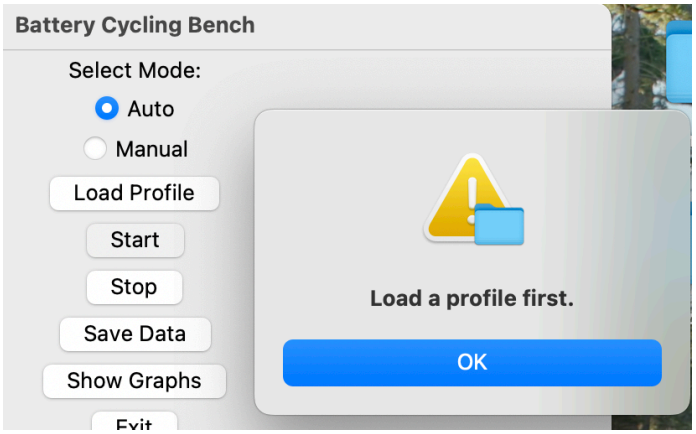

Figure 23 : Message si aucun profile n'a été chargé avant l'acquisition

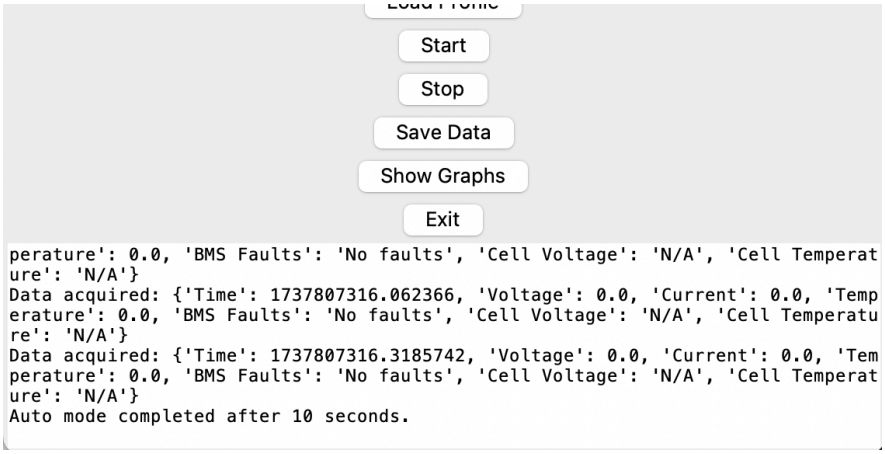

Figure 24 : Affichage de l'acquisition des données

4. Visualisation des données

- Graphiques de tension, courant, température et autres paramètres.

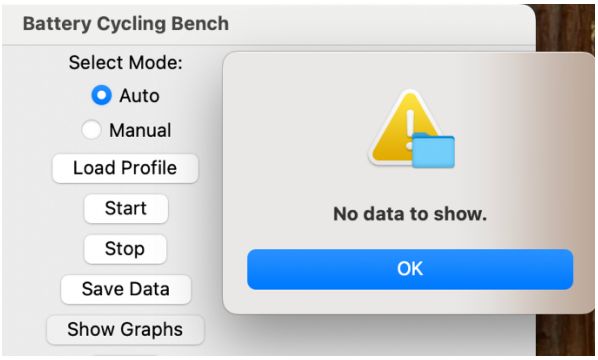

Figure 25 : Message si aucune donnée ne peut être affichée

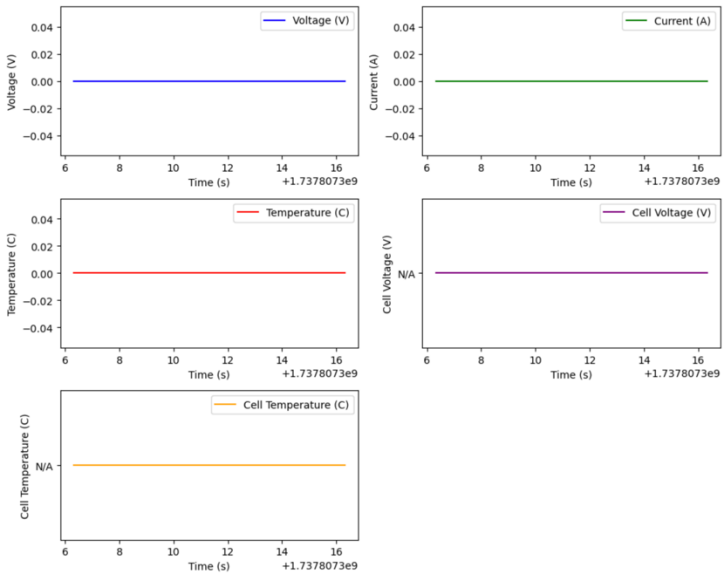

Figure 26 : Affichage des données

5. Exportation des données

- Enregistrement des mesures dans un fichier CSV.

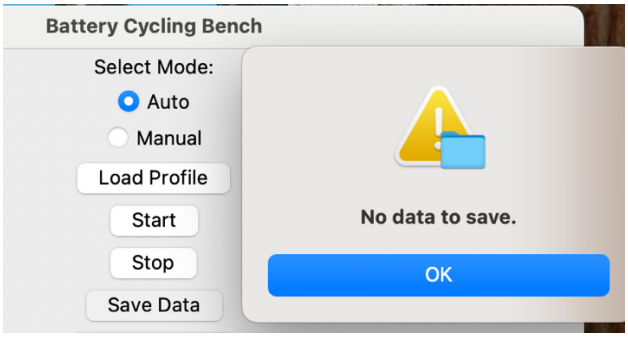

Figure 27 : Message si aucune donnée ne peut être sauvegardée

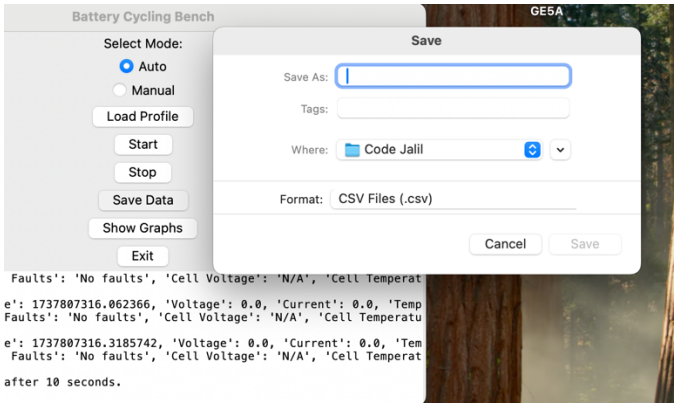

Figure 28 : Sauvegarde des données une fois acquises

2.3. Description des fonctions

Classe « BatteryCyclingApp » :

a) Attributs principaux :

- « data_log » : liste contenant les mesures collectées.
- « is_running » : booléen indiquant si le cycle est actif.
- « profile_dat » : liste des paramètres chargés depuis un profil CSV.
- « mode » : mode de fonctionnement (manuel ou automatique).

```
# Variables
self.data_log = []
self.is_running = False
self.profile_data = []
self.mode = tk.StringVar(value="auto")
```

Figure 29 : Variables principales

b) Méthodes principales :

- « __init__(self, root) »

- Initialise la fenêtre principale et les composants de l'IHM.

```python
def __init__(self, root):
    self.root = root
    self.root.title("Battery Cycling Bench")

    # Variables
    self.data_log = []
    self.is_running = False
    self.profile_data = []
    self.mode = tk.StringVar(value="auto")

    # UI Components
    self.create_widgets()
```

Figure 30 : Fonction d'initialisation de l'IHM

- « create_widgets(self) »

- Crée et place les widgets dans l'IHM.

```python
def create_widgets(self):
    # Mode Selection
    tk.Label(self.root, text="Select Mode:").pack()
    tk.Radiobutton(self.root, text="Auto", variable=self.mode, value="auto").pack()
    tk.Radiobutton(self.root, text="Manual", variable=self.mode, value="manual").pack()

    # Buttons
    tk.Button(self.root, text="Load Profile", command=self.load_profile).pack()
    tk.Button(self.root, text="Start", command=self.start_cycling).pack()
    tk.Button(self.root, text="Stop", command=self.stop_cycling).pack()
    tk.Button(self.root, text="Save Data", command=self.save_data).pack()
    tk.Button(self.root, text="Show Graphs", command=self.show_graphs).pack()
    tk.Button(self.root, text="Exit", command=self.root.quit).pack()

    # Log Display
    self.log_text = tk.Text(self.root, state='disabled', height=10)
    self.log_text.pack()
```

Figure 31 : Fonction de création de l'affichage opérationnel de l'IHM

- « load_profile(self) »

- Permet de charger un fichier CSV contenant les paramètres des cycles.
- Les données sont stockées dans « profile_data » sous forme de dictionnaire.

```python
def load_profile(self):
    file_path = filedialog.askopenfilename(filetypes=[("CSV Files", "*.csv")])
    if not file_path:
        return

    with open(file_path, newline='') as csvfile:
        reader = csv.DictReader(csvfile)
        self.profile_data = [row for row in reader]
```

Figure 32 : Fonction de récupération de fichiers .csv

- « start_cycling(self)»

- Démarre le processus de cyclage dans un thread séparé (mode automatique ou manuel).

```
def start_cycling(self):
    if not self.profile_data:
        messagebox.showwarning("Warning", "Load a profile first.")
        return

    self.is_running = True
    if self.mode.get() == "auto":
        self.log_message("Starting auto mode...")
        Thread(target=self.run_auto_mode).start()
    elif self.mode.get() == "manual":
        self.log_message("Starting manual mode...")
        Thread(target=self.run_manual_mode).start()
```

Figure 33 : Fonction de gestion du bouton "Start"

- « stop_cycling(self)»

- Arrête le processus de cyclage.

```
def stop_cycling(self):
    self.is_running = False
    self.log_message("Cycling stopped.")
```

Figure 34 : Fonction de gestion du bouton "Stop"

- « run_auto_mode(self) »

- Simule un mode automatique avec acquisition continue des données pendant 10 secondes.

```
def run_auto_mode(self):
    start_time = time.time()
    while self.is_running and (time.time() - start_time) < 10:
        self.acquire_data()
        time.sleep(0.25)
    self.is_running = False
    self.log_message("Auto mode completed after 10 seconds.")
    self.show_graphs()
```

Figure 35 : Fonction d'acquisition des données en mode "Auto"

- « run_manual_mode(self) »

- Exécute un cycle manuel basé sur les paramètres chargés depuis le profil.

```
def run_manual_mode(self):
    for profile in self.profile_data:
        if not self.is_running:
            break
        self.acquire_data(profile)
        time.sleep(0.25)
```

Figure 36 : Fonction d'acquisition des données en mode "Manual"

- “acquire_data(self, profile=None)”

- Simule l'acquisition de données (temps, tension, courant, etc.).

```python
def acquire_data(self, profile=None):
    # Simulated data acquisition
    if profile:
        data = {
            "Time": time.time(),
            "Voltage": float(profile["Voltage"]),
            "Current": float(profile["Current"]),
            "Temperature": float(profile["Temperature"]),
            "BMS Faults": profile.get("BMS Faults", "No faults"),
            "Cell Voltage": profile.get("Cell Voltage", "N/A"),
            "Cell Temperature": profile.get("Cell Temperature", "N/A"),
        }
    else:
        data = {
            "Time": time.time(),
            "Voltage": 0.0,
            "Current": 0.0,
            "Temperature": 0.0,
            "BMS Faults": "No faults",
            "Cell Voltage": "N/A",
            "Cell Temperature": "N/A",
        }

    self.data_log.append(data)
    self.log_message(f"Data acquired: {data}")
```

Figure 37 : Fonction d'acquisition des données

- « log_message(self, message) »

- Affiche des messages dans la fenêtre de log.

```python
def log_message(self, message):
    self.log_text.config(state='normal')
    self.log_text.insert(tk.END, f"{message}\n")
    self.log_text.config(state='disabled')
    self.log_text.see(tk.END)
```

Figure 38 : Fonction d'écriture dans la fenêtre

- « save_data(self) »

- Enregistre les mesures collectées dans un fichier CSV.

```python
def save_data(self):
    if not self.data_log:
        messagebox.showwarning("Warning", "No data to save.")
        return

    file_path = filedialog.asksaveasfilename(defaultextension=".csv", filetypes=[("CSV Files", "*.csv")])
    if not file_path:
        return

    with open(file_path, mode='w', newline='') as csvfile:
        fieldnames = ["Time", "Voltage", "Current", "Temperature", "BMS Faults", "Cell Voltage", "Cell Temperature"]
        writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
        writer.writeheader()
        writer.writerows(self.data_log)

    messagebox.showinfo("Data Saved", "Data successfully saved.")
```

Figure 39 : Fonction de sauvegarde des données

- « show_graphs(self) »

- Affiche les graphiques des différents paramètres mesurés.

```python
def show_graphs(self):
    if not self.data_log:
        messagebox.showwarning("Warning", "No data to show.")
        return

    times = [data["Time"] for data in self.data_log]
    voltages = [data["Voltage"] for data in self.data_log]
    currents = [data["Current"] for data in self.data_log]
    temperatures = [data["Temperature"] for data in self.data_log]
    cell_voltages = [data["Cell Voltage"] for data in self.data_log]
    cell_temperatures = [data["Cell Temperature"] for data in self.data_log]

    plt.figure(figsize=(10, 8))

    # Voltage graph
    plt.subplot(3, 2, 1)
    plt.plot(times, voltages, label="Voltage (V)", color='blue')
    plt.xlabel("Time (s)")
    plt.ylabel("Voltage (V)")
    plt.legend()

    # Current graph
    plt.subplot(3, 2, 2)
    plt.plot(times, currents, label="Current (A)", color='green')
    plt.xlabel("Time (s)")
    plt.ylabel("Current (A)")
    plt.legend()

    # Temperature graph
    plt.subplot(3, 2, 3)
    plt.plot(times, temperatures, label="Temperature (C)", color='red')
    plt.xlabel("Time (s)")
    plt.ylabel("Temperature (C)")
    plt.legend()

    # Cell Voltage graph
    plt.subplot(3, 2, 4)
    plt.plot(times, cell_voltages, label="Cell Voltage (V)", color='purple')
    plt.xlabel("Time (s)")
    plt.ylabel("Cell Voltage (V)")
    plt.legend()

    # Cell Temperature graph
    plt.subplot(3, 2, 5)
    plt.plot(times, cell_temperatures, label="Cell Temperature (C)", color='orange')
    plt.xlabel("Time (s)")
    plt.ylabel("Cell Temperature (C)")
    plt.legend()

    plt.tight_layout()
    plt.show()
```

Figure 40 : Fonction d'affichage des données acquises

2.4. Fonctionnement de l'IHM

1. L'utilisateur choisit le mode d'opération (« Auto » ou « Manual »).
2. En mode manuel, un fichier CSV de profil doit être chargé.
3. Le processus de cyclage est démarré (« Start »).
4. Les données sont acquises et affichées dans le log.
5. L'utilisateur peut arrêter le processus (« Stop ») à tout moment.
6. Les données peuvent être visualisées et/ou sauvegardées.

2.5. Aspects techniques

- Threads : Les modes d'acquisition fonctionnent dans des threads séparés pour maintenir la réactivité de l'IHM.
- Visualisation : « matplotlib » permet une visualisation claire et segmentée des données collectées.
- « Gestion des fichiers » : Utilisation de « csv.DictReader » et « csv.DictWriter » pour manipuler les profils et sauvegarder les données.

2.6. Points d'amélioration possibles

1. Décorréler les fichiers .csv chargés (les profils) des données acquises. (Fonction « run_manual_mode(self) »)
2. Permettre la sélection des graphiques par l'utilisateur en fin d'acquisition (Fonction « show_graphs(self) »)
3. Permettre à l'IHM de communiquer avec les différents autres appareils du système (L'alimentation bidirectionnelle et la carte d'acquisition)
4. Ajouter des contrôles avancés pour définir les durées et les paramètres de cyclage directement depuis l'IHM.
5. Implémenter un bouton pour pauser/reprendre le cycle.
6. Optimiser la gestion des threads pour améliorer la stabilité lors de longues sessions.

Conclusion :

Il vous a donc été présenté, dans le détail, la technique qui a permis le développement de ces parties du projet plus large de conception d'un banc de test de batteries. La communication avec l'alimentation est donc totalement opérationnelle. L'exécution des cycles pourra être développé à la suite en se basant sur les fonctions qui ont été codées. Du côté de l'IHM, plus de travail devra être fourni. En effet, son développement finale (la suite de ce qui a déjà été fait) nécessite la validation de plusieurs autres blocs cruciaux du banc de test.

Tous les documents qui ont permis ce développement sont répertoriés dans la bibliographie du rapport de stage.

Annexe

Annexe 1 : Code de communication ETH avec l'alimentation bidirectionnelle :

```
import socket
import time

# Paramètres de connexion
IP_ADDRESS = "192.168.0.3"
PORT = 5025
NUM_CYCLES = 5 # Nombre de cycles charge/décharge

# Fonctions utilitaires
def send_command(sock, command):
  """Envoie une commande SCPI via une connexion existante."""
  try:
    if sock.fileno() == -1: # Vérifie si le socket est actif
      print("Le socket est fermé. Reconnexion...")
      sock = reconnect(sock)

    command = command.strip() + "\n"
    sock.sendall(command.encode())
    time.sleep(1) # Augmentez le délai à 1 seconde
    response = sock.recv(1024).decode().strip()
    print(f"Commande: {command.strip()}\nRéponse: {response}")
    return response
  except BrokenPipeError:
    print("Connexion rompue. Tentative de reconnexion...")
    raise
  except Exception as e:
    print(f"Erreur lors de l'envoi de la commande {command.strip()}: {e}")
    raise

def connect_to_device():
  """Établit une connexion au dispositif."""
  try:
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.settimeout(10)
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_KEEPALIVE, 1) # Active TCP Keep
                - Alive
    sock.connect((IP_ADDRESS, PORT))
    print("Connecté à l'alimentation.")
    return sock
  except Exception as e:
    print(f"Erreur lors de la connexion : {e}")
    raise

def reconnect(sock):
  """Rétablit une connexion en cas de perte."""
  try:
    sock.close()
```

```
    except:
        pass
    return connect_to_device()

def log_device_error(sock):
    """Récupère les erreurs de l'appareil."""
    try:
        error_log = send_command(sock, "SYST:ERR?")
        print(f"Log de l'appareil : {error_log}")
    except Exception as e:
        print(f"Impossible de lire le log de l'appareil : {e}")

def initialize_device(sock):
    """Initialise le dispositif et active le contrôle à distance."""
    # Identification de l'appareil
    response = send_command(sock, "*IDN?")
    if not response:
        raise Exception("Aucune réponse de l'appareil. Vérifiez la connexion.")

    # Activation du contrôle à distance et configuration
    send_command(sock, "SYST:LOCK ON")
    send_command(sock, "SYST:TIMEOUT 0")  # Désactivation du timeout SCPI
    send_command(sock, "*CLS")  # Efface les erreurs précédentes
    send_command(sock, "*RST")  # Réinitialisation de l'appareil

def cycle_charge_discharge(sock):
    """Effectue les cycles de charge et décharge."""
    for cycle in range(1, NUM_CYCLES + 1):
        print(f"\nDébut du cycle {cycle}/{NUM_CYCLES}")

        # Étape 1 : Charge
        print("Démarrage de la charge...")
        try:
            time.sleep(0.5)  # Délai avant de configurer la tension
            send_command(sock, "VOLT 10")
            time.sleep(10)  # Temps de charge (exemple : 60 secondes)

        except BrokenPipeError:
            print("Reconnexion nécessaire pendant la charge.")
            sock = reconnect(sock)
            log_device_error(sock)
            continue

        # Étape 2 : Décharge
        #print("Démarrage de la décharge...")
        #try:
            #send_command(sock, "SINK:CURR 2")  # Exemple : Courant de décharge négatif
            #time.sleep(10)  # Temps de décharge (exemple : 60 secondes)
        #except BrokenPipeError:
            #print("Reconnexion nécessaire pendant la décharge.")
```

```
        #sock = reconnect(sock)
        #log_device_error(sock)
        #continue

    print("\nTous les cycles sont terminés.")
    send_command(sock, "OUTP:STAT OFF") # Désactive la sortie

# Script principal
try:
    sock = connect_to_device()
    initialize_device(sock)
    send_command(sock, "OUTP ON")
    cycle_charge_discharge(sock)
except BrokenPipeError:
    print("Erreur critique : Connexion rompue. Tentative de réinitialisation...")
    sock = reconnect(sock)
except Exception as e:
    print(f"Erreur critique : {e}")
finally:
    if sock:
        try:
            send_command(sock, "OUTP OFF")
            send_command(sock, "OUTP:STAT OFF") # Assurez
                    - vous que l'alimentation est arrêtée

        except:
            pass
        sock.close()
        print("Connexion fermée.")
```

Annexe 2 : Code de l'IHM :

```
import tkinter as tk
from tkinter import filedialog, messagebox
import csv
import time
from threading import Thread
import matplotlib.pyplot as plt

class BatteryCyclingApp:
    def __init__(self, root):
        self.root = root
        self.root.title("Battery Cycling Bench")

        # Variables
        self.data_log = []
        self.is_running = False
        self.profile_data = []
        self.mode = tk.StringVar(value="auto")
```

```
    # UI Components
    self.create_widgets()

def create_widgets(self):
    # Mode Selection
    tk.Label(self.root, text = "Select Mode:").pack()
    tk.Radiobutton(self.root, text = "Auto", variable = self.mode, value = "auto").pack()
    tk.Radiobutton(self.root, text = "Manual", variable = self.mode, value
                = "manual").pack()

    # Buttons
    tk.Button(self.root, text = "Load Profile", command = self.load_profile).pack()
    tk.Button(self.root, text = "Start", command = self.start_cycling).pack()
    tk.Button(self.root, text = "Stop", command = self.stop_cycling).pack()
    tk.Button(self.root, text = "Save Data", command = self.save_data).pack()
    tk.Button(self.root, text = "Show Graphs", command = self.show_graphs).pack()
    tk.Button(self.root, text = "Exit", command = self.root.quit).pack()

    # Log Display
    self.log_text = tk.Text(self.root, state = 'disabled', height = 10)
    self.log_text.pack()

def load_profile(self):
    file_path = filedialog.askopenfilename(filetypes = [("CSV Files", "*.csv")])
    if not file_path:
        return

    with open(file_path, newline = '') as csvfile:
        reader = csv.DictReader(csvfile)
        self.profile_data = [row for row in reader]

    messagebox.showinfo("Profile Loaded", "Profile successfully loaded.")

def start_cycling(self):
    if not self.profile_data:
        messagebox.showwarning("Warning", "Load a profile first.")
        return

    self.is_running = True
    if self.mode.get() == "auto":
        self.log_message("Starting auto mode...")
        Thread(target = self.run_auto_mode).start()
    elif self.mode.get() == "manual":
        self.log_message("Starting manual mode...")
        Thread(target = self.run_manual_mode).start()

def stop_cycling(self):
    self.is_running = False
    self.log_message("Cycling stopped.")
```

```
def run_auto_mode(self):
    start_time = time.time()
    while self.is_running and (time.time() - start_time) < 10:
        self.acquire_data()
        time.sleep(0.25)
    self.is_running = False
    self.log_message("Auto mode completed after 10 seconds.")
    self.show_graphs()

def run_manual_mode(self):
    for profile in self.profile_data:
        if not self.is_running:
            break
        self.acquire_data(profile)
        time.sleep(0.25)

def acquire_data(self, profile = None):
    # Simulated data acquisition
    if profile:
        data = {
            "Time": time.time(),
            "Voltage": float(profile["Voltage"]),
            "Current": float(profile["Current"]),
            "Temperature": float(profile["Temperature"]),
            "BMS Faults": profile.get("BMS Faults", "No faults"),
            "Cell Voltage": profile.get("Cell Voltage", "N/A"),
            "Cell Temperature": profile.get("Cell Temperature", "N/A"),
        }
    else:
        data = {
            "Time": time.time(),
            "Voltage": 0.0,
            "Current": 0.0,
            "Temperature": 0.0,
            "BMS Faults": "No faults",
            "Cell Voltage": "N/A",
            "Cell Temperature": "N/A",
        }

    self.data_log.append(data)
    self.log_message(f"Data acquired: {data}")

def log_message(self, message):
    self.log_text.config(state = 'normal')
    self.log_text.insert(tk.END, f"{message}\n")
    self.log_text.config(state = 'disabled')
    self.log_text.see(tk.END)

def save_data(self):
```

```
        if not self.data_log:
            messagebox.showwarning("Warning","No data to save.")
            return

        file_path = filedialog.asksaveasfilename(defaultextension=".csv",filetypes
                    =[("CSV Files","*.csv")])
        if not file_path:
            return

        with open(file_path,mode='w',newline='') as csvfile:
            fieldnames
= ["Time","Voltage","Current","Temperature","BMS Faults","Cell Voltage","Cell Temperature"]
            writer = csv.DictWriter(csvfile,fieldnames=fieldnames)
            writer.writeheader()
            writer.writerows(self.data_log)

        messagebox.showinfo("Data Saved","Data successfully saved.")

    def show_graphs(self):
        if not self.data_log:
            messagebox.showwarning("Warning","No data to show.")
            return

        times = [data["Time"] for data in self.data_log]
        voltages = [data["Voltage"] for data in self.data_log]
        currents = [data["Current"] for data in self.data_log]
        temperatures = [data["Temperature"] for data in self.data_log]
        cell_voltages = [data["Cell Voltage"] for data in self.data_log]
        cell_temperatures = [data["Cell Temperature"] for data in self.data_log]

        plt.figure(figsize=(10,8))

        # Voltage graph
        plt.subplot(3,2,1)
        plt.plot(times,voltages,label="Voltage (V)",color='blue')
        plt.xlabel("Time (s)")
        plt.ylabel("Voltage (V)")
        plt.legend()

        # Current graph
        plt.subplot(3,2,2)
        plt.plot(times,currents,label="Current (A)",color='green')
        plt.xlabel("Time (s)")
        plt.ylabel("Current (A)")
        plt.legend()

        # Temperature graph
        plt.subplot(3,2,3)
        plt.plot(times,temperatures,label="Temperature (C)",color='red')
        plt.xlabel("Time (s)")
```

```
        plt.ylabel("Temperature (C)")
        plt.legend()

        # Cell Voltage graph
        plt.subplot(3, 2, 4)
        plt.plot(times, cell_voltages, label = "Cell Voltage (V)", color = 'purple')
        plt.xlabel("Time (s)")
        plt.ylabel("Cell Voltage (V)")
        plt.legend()

        # Cell Temperature graph
        plt.subplot(3, 2, 5)
        plt.plot(times, cell_temperatures, label = "Cell Temperature (C)", color = 'orange')
        plt.xlabel("Time (s)")
        plt.ylabel("Cell Temperature (C)")
        plt.legend()

        plt.tight_layout()
        plt.show()

if __name__ == "__main__":
    root = tk.Tk()
    app = BatteryCyclingApp(root)
    root.mainloop()
```