MU4MEN01 – Projet d'Optimisation

AKIL Adam, BUCLET Zeca, NOCHÉ Kévin

10 janvier 2025

Table des matières

Introduction 1
La fonction `Simulation()` 1
Sans Batterie 1
Avec Batterie 2
Analyse des sous-stations et qualité du système 4
Monte-Carlo 6
Conclusion Monte-Carlo 11
Algorithme NSGA-II 11
Conclusion 13

Introduction

Dans ce présent document, nous allons étudier un modèle représentant un train, sa consommation, sa batterie et comment optimiser deux choses :

— La capacité de ladite batterie.

— La chute de tension maximale aux bornes du train, sachant que nous ne devons pas atteindre moins de 500 V aux bornes du train.

Le modèle auquel nous nous intéressons a été codé en python (le code est joint au document), et prend en compte, de façon non-exhaustive, les déplacements du train, sa vitesse, son accélération, sa puissance consommée et la puissance de la LAC (ligne aérienne de contact). Le fichier `marche.txt`, qui nous a été remis dès le début du projet, indique les déplacements du train en fonction du temps. De rapides et simples calculs nous permettent de trouver la vitesse du train, puis son accélération, primordiales pour la suite du projet.

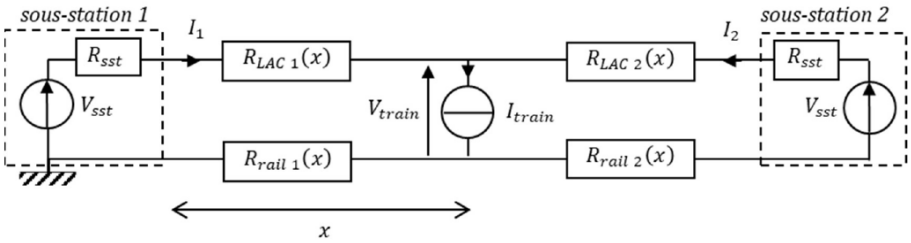

FIGURE 1 : Schéma du modèle.

La fonction `Simulation()`

Sans Batterie

Lors de l'analyse du mouvement du train, les graphiques montrent les interactions entre la position, la vitesse et l'accélération. Une phase d'accélération entraîne une augmentation plus marquée des courbes de position et de vitesse. À l'inverse, pendant le freinage, la position évolue lentement tandis que la vitesse et l'accélération diminuent progressivement jusqu'à atteindre zéro, tant que l'accélération reste nulle.

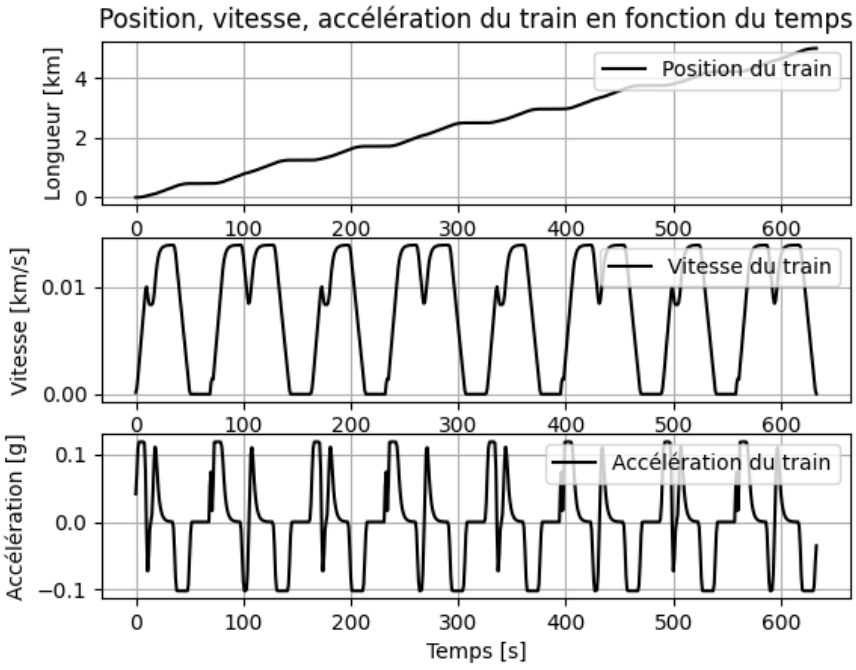

FIGURE 2 : Position, Vitesse et Accélération du train.

Pour calculer ces variations, la fonction `np.gradient` a été utilisée.

Ensuite, la puissance du train a été déterminée en intégrant les forces résistives, motrices et mécaniques, en tenant compte d'un rendement global de 80% pour la chaîne de conversion.

```
# F_resistive.
F_resistive = (A_0+A_1*M)+(B_0+B_1*M)*v+(C_0+C_1*M)*v**2

# F_motrice.
F_motrice = M*a+M*g*np.sin(alpha)+F_resistive

# P_mecanique.
P_mecanique = F_motrice*v

# Puissance totale consommée par le train: P_train.
P_train = np.zeros(len(P_mecanique))
for k in range(len(P_mecanique)):
    if P_mecanique[k] > 0:
        P_train[k] = P_mecanique[k]/rend+SysBor # On consomme l'énergie.
    else:
        P_train[k] = P_mecanique[k]*rend+SysBor # On produit l'énergie.
```

FIGURE 3

Avec Batterie

L'introduction d'une batterie (rendement de 90%, capacité de 10 kWh et tension de seuil de 280 kV) modifie les dynamiques observées. Les graphiques indiquent que les relations entre la puissance, la tension et le courant restent cohérentes. Durant les phases d'accélération, la batterie se décharge, tandis qu'elle se recharge lors du freinage.

Concernant la gestion de la batterie, on a suivi la logique (indiqué figure 4) :

```
for k in range(1, len(t)):
    if P_train[k] < 0: # Le train freine.
        Bat_E[k] = Bat_E[k-1]-P_train[k] * Bat_rend # On récupère l'énergie.
        P_train[k] = 0 # Le train ne consomme plus d'énergie.
        if Bat_E[k] > 3600*Bat_cap: # Dépasse-t-on les limites de la batterie?
            Diff = Bat_E[k] - Bat_cap*3600 # On dissipe dans le rhéostat.
            Bat_E[k] = Bat_cap*3600 # La batterie est pleine.
            Rheo_P[k] = Diff # On dissipe dans le rhéostat.
    elif P_train[k] >= seuil: # Si le train demande de l'énergie.
        Bat_E[k] = Bat_E[k-1] - (P_train[k]-seuil) # On consomme l'énergie.
        P_train[k] = P_train[k] - seuil * Bat_rend # On consomme l'énergie.
        if Bat_E[k] < 0: # Dépasse-t-on les limites de la batterie?
            Diff = -Bat_E[k] # On dissipe dans le rhéostat.
            Bat_E[k] = 0 # La batterie est vide.
            P_train[k] += Diff # On dissipe dans le rhéostat.
    else:
        Bat_E[k] = Bat_E[k-1] # La batterie conserve son énergie.
```

FIGURE 4 : Gestion de la batterie.

Le système globale peut-être schématisé comme le modèle ci-dessous :

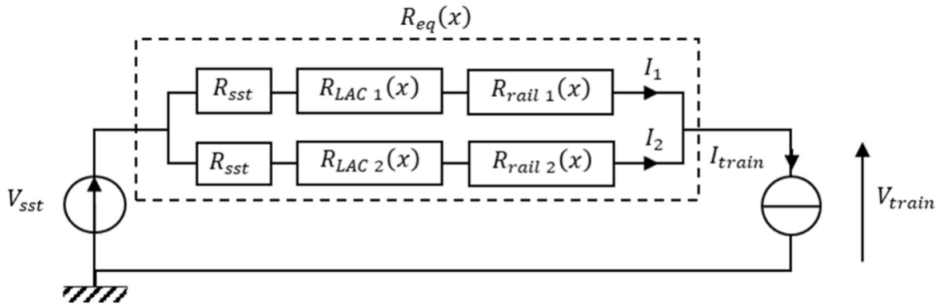

Figure 5 : Modèle équivalent du système.

FIGURE 5 : Schéma des résistances du circuit.

$$R_{\text{LAC1}} = \rho_{\text{LAC}} x$$
$$R_{\text{LAC2}} = \rho_{\text{LAC}} (x_{\text{Tot}} - x)$$
$$R_{\text{rail1}} = x\rho_{\text{rail}}$$
$$R_{\text{rail2}} = (x_{\text{Tot}} - x)\rho_{\text{rail}}$$
$$R_{\text{eq}} = \frac{(R_{\text{SST}} + R_{\text{LAC1}} + R_{\text{rail1}})(R_{\text{SST}} + R_{\text{LAC2}} + R_{\text{rail2}})}{2R_{\text{SST}} + R_{\text{LAC1}} + R_{\text{LAC2}} + R_{\text{rail1}} + R_{\text{rail2}}}$$

Cela nous permet ainsi de calculer la tension et le courant du train.

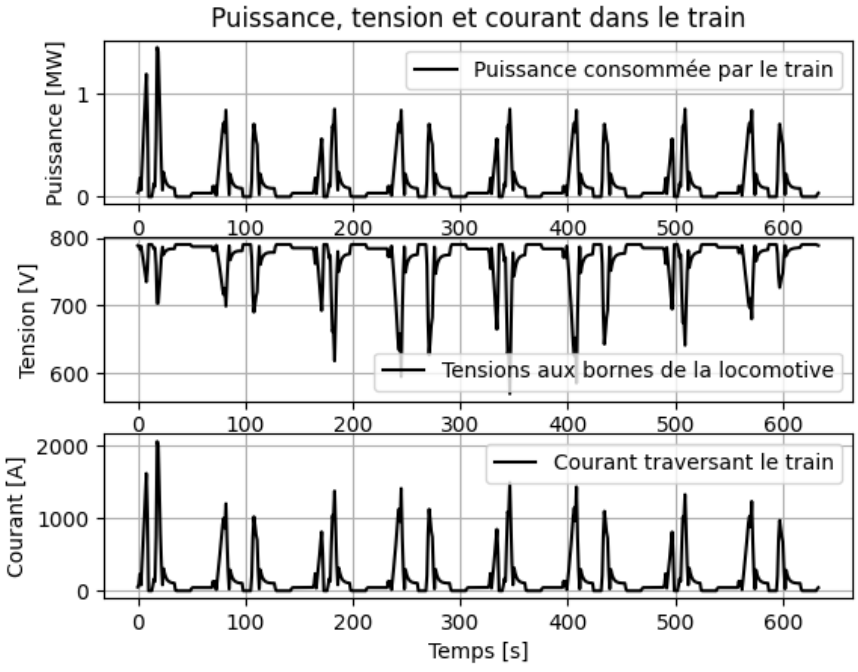

FIGURE 6 : Puissance, tension et courant du train.

On observe ainsi que les relations entre la puissance, la tension et le courant sont respectées. Lors des phases d'accélération ou de freinage on a une variation de ces éléments.

Concernant la batterie, celle-ci réagit également comme nous l'avions prévu, à savoir que lors du freinage celle-ci se recharge, puis lors des phases d'accélération elle se décharge.

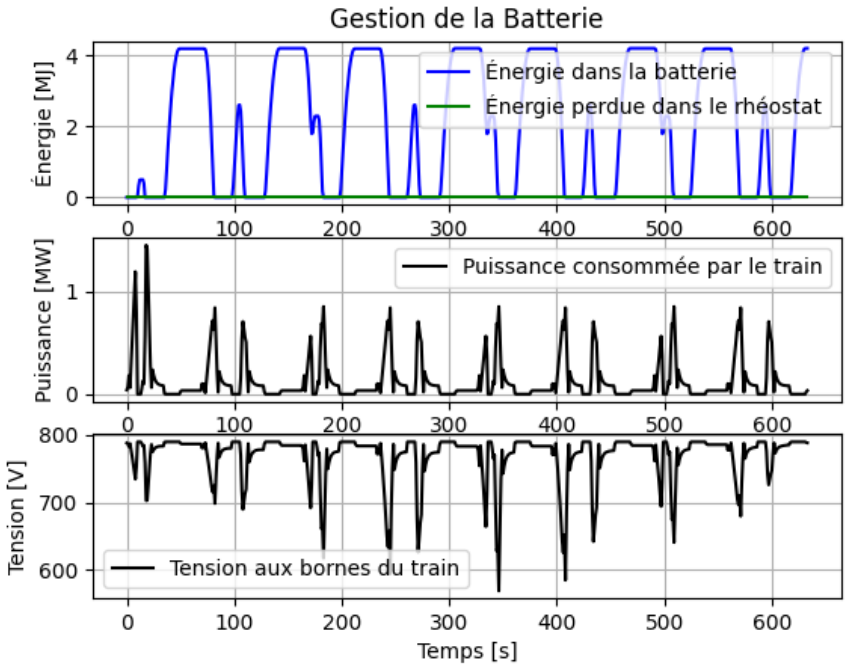

FIGURE 7 : Gestion de la batterie.

Analyse des sous-stations et qualité du système

Les graphiques montrent que le courant et la puissance dans les LAC des sous-stations sont équilibrés. On constate également que, lorsque le train s'approche de la sous-station 2, la puissance et le courant fournis par celle-ci augmentent proportionnellement (figure 8).

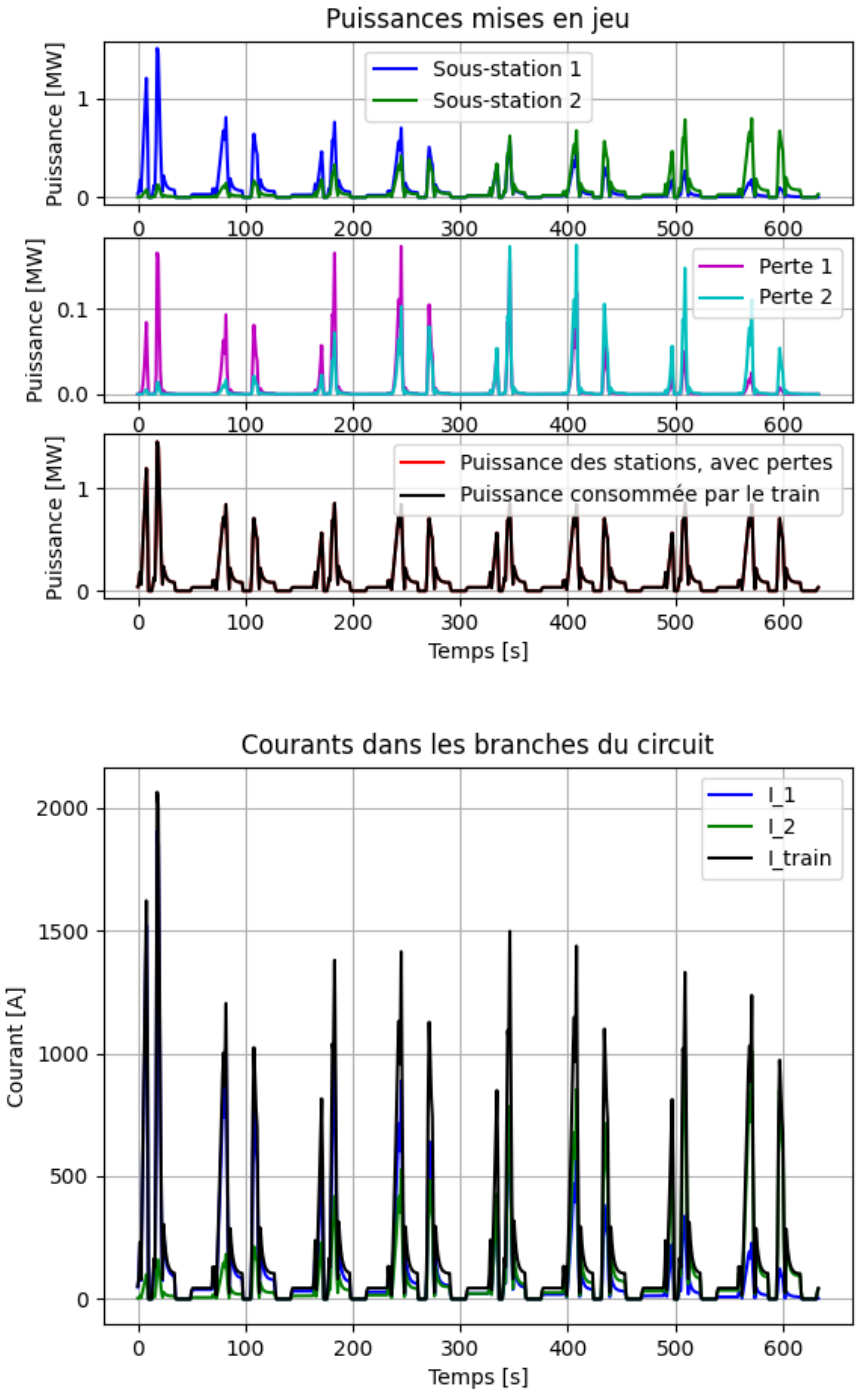

FIGURE 8 : Puissances et courants mis en jeu.

Un indicateur clé de performance, la chute de tension maximale, est utilisé pour évaluer la qualité du système. Cet indicateur pourrait être optimisé en choisissant une batterie et une tension de seuil mieux adaptées.

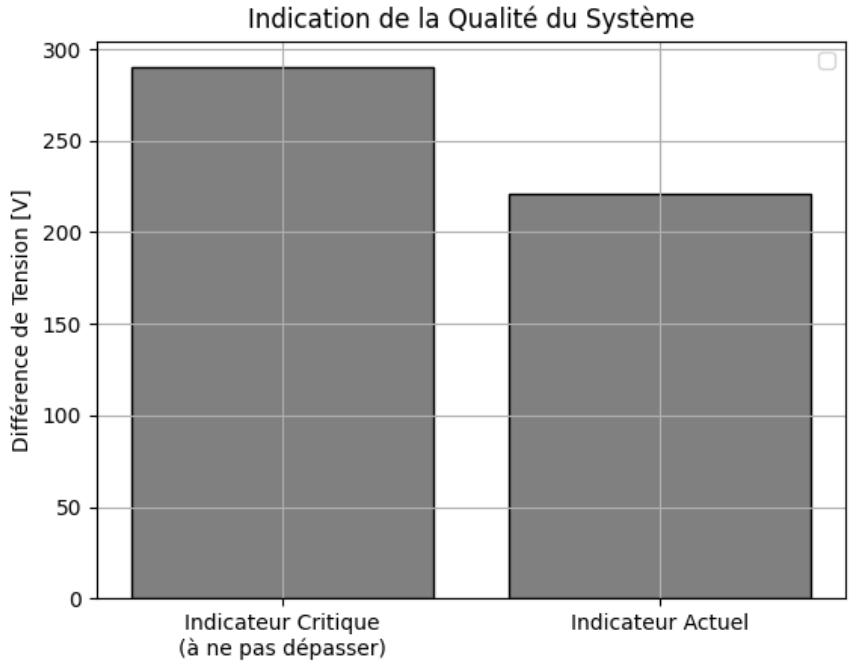

FIGURE 9 : Chute de tension (indicateur de qualité) du système.

Monte-Carlo

Pour optimiser notre système, nous avons d'abord utilisé la méthode Monte-Carlo. Cette méthode est facile à implémenter et repose uniquement sur une utilisation brute de la force de calcul.

Comme décrit dans l'énoncé, on cherche à échantillonner l'espace des solutions avec un grand nombre de points puis de calculer leurs performances. On place ensuite ces points dans l'espace des solutions et déterminons les solutions non dominées et finalement la solution optimale.

Echantillonnage

Avec la fonction *np.linespace()* on commence par créer deux tableaux de 1000 valeurs également espacées :

- **Capa** (capacité de la batterie) allant de 0 à 14 kW.
- **Seuil** (Puissance seuil maximum du train) allant de 0 à 1 MW.

Nous avons choisi ces valeurs par rapport au PDF de complément.

Affichage des solutions

On choisit ensuite 1000 couples (**Capa**, **Seuil**) de façon aléatoire avec la fonction *random.choice()*. En plottant cela dans l'espace des solutions nous obtenons le graphique suivant :

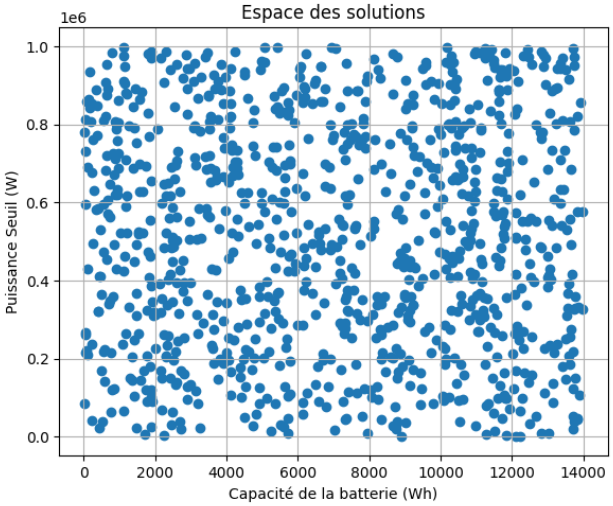

FIGURE 10 : Espace des solutions.

En reportant cela dans l'espace des objectifs, nous obtenons :

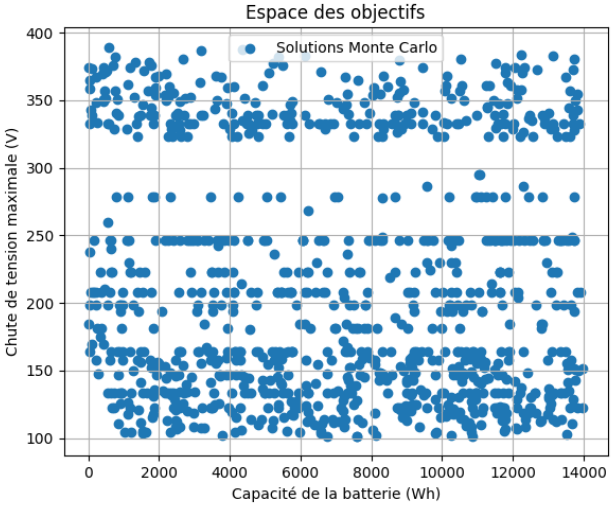

FIGURE 11 : Espace des objectifs.

Nous avons ensuite crée une fonction qui prend en argument une population et retourne les solutions non dominées. Cette fonction s'appelle *non_dominant_sort(*pop*)* et est disponible dans notre code.

Voici nos résultats avec les solutions non dominées en rouge :

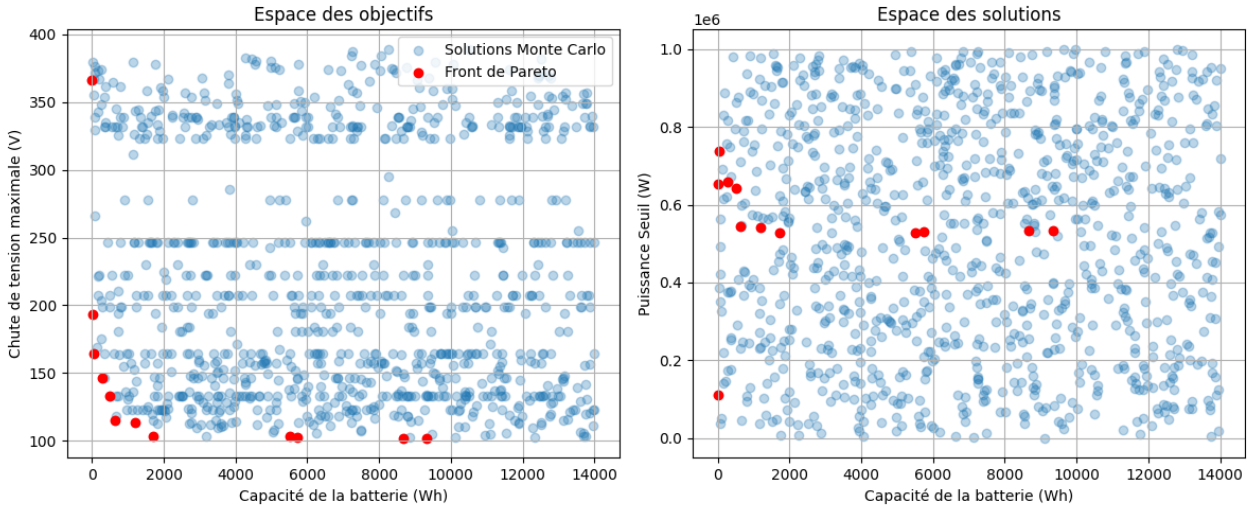

FIGURE 12 : Espace des objectifs avec solutions non dominées.

Enfin, nous déterminons la meilleure solution avec la fonction *choisir_meilleur_point()* qui prend en paramètre le front de pareto, ainsi que le poids que nous accordons à chacun des paramètres à optimiser (à savoir la chute de tension et la capacité de la batterie).

Sur le graphique précédant, nous pouvons voir que la chute de tension possède une grande plage de variation sur les meilleures solutions tandis que la capacité reste surtout entre 0 et 2000. Nous accordons donc un poids plus important à la chute de tension. Nous nous retrouvons enfin avec le graphique ci dessous, avec en vert la solution la plus optimale et ses spécificités en note sous les graphiques :

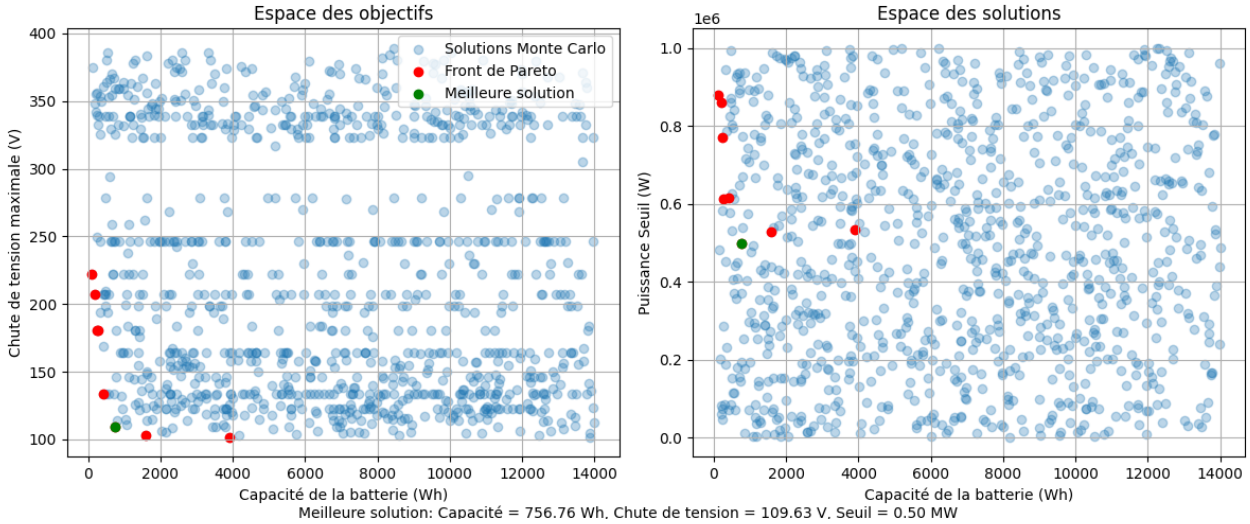

FIGURE 13 : Espace des objectifs avec solutions non dominées et meilleure solution.

Résultats

En simulant plusieurs fois, nous obtenons des résultats qui varient légèrement à cause de la répartition aléatoire des points. Ci-dessous se trouvent 3 graphiques avec toujours 1000 couples échantillons.

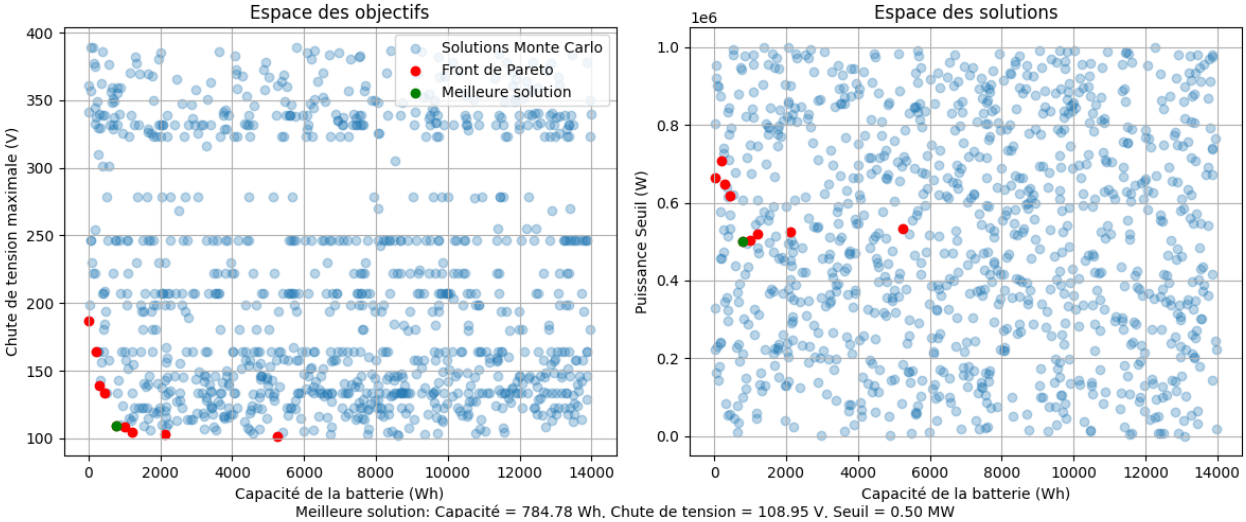

FIGURE 14 : Espace des objectifs avec solutions non dominées et meilleure solution.

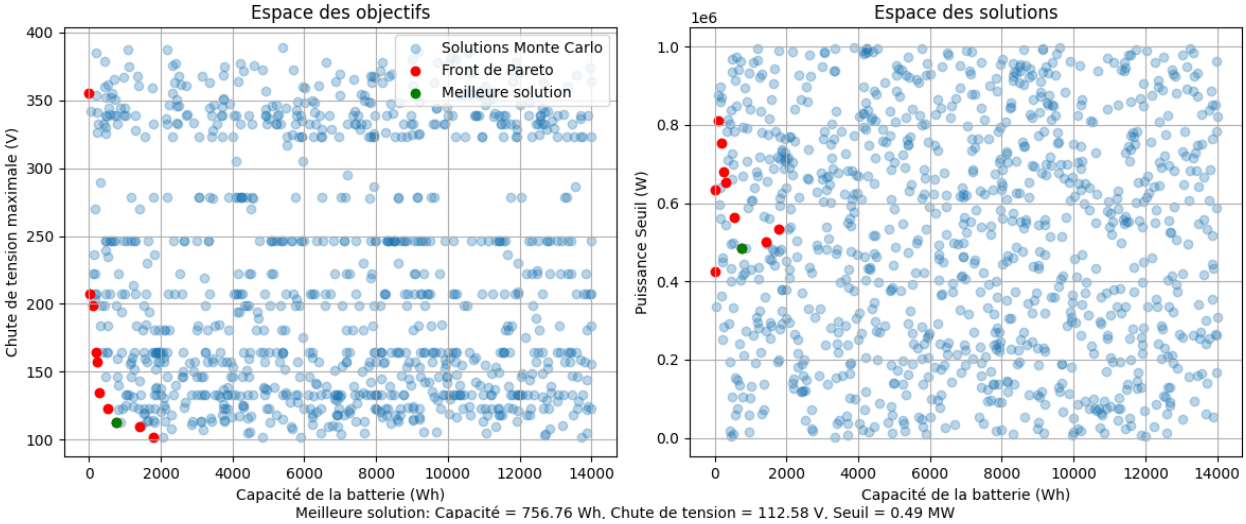

FIGURE 15 : Espace des objectifs avec solutions non dominées et meilleure solution.

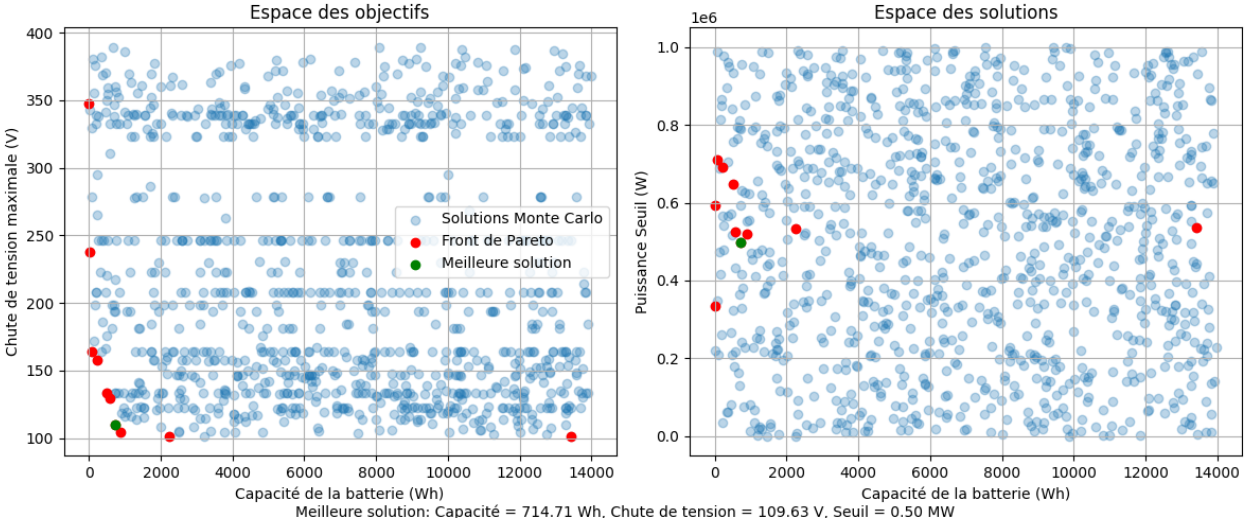

FIGURE 16 : Espace des objectifs avec solutions non dominées et meilleure solution.

En augmentant le nombre d'échantillons, nous pouvons avoir une meilleure précision. Ci-dessous sont les résultats avec $N = 5000$ couples.

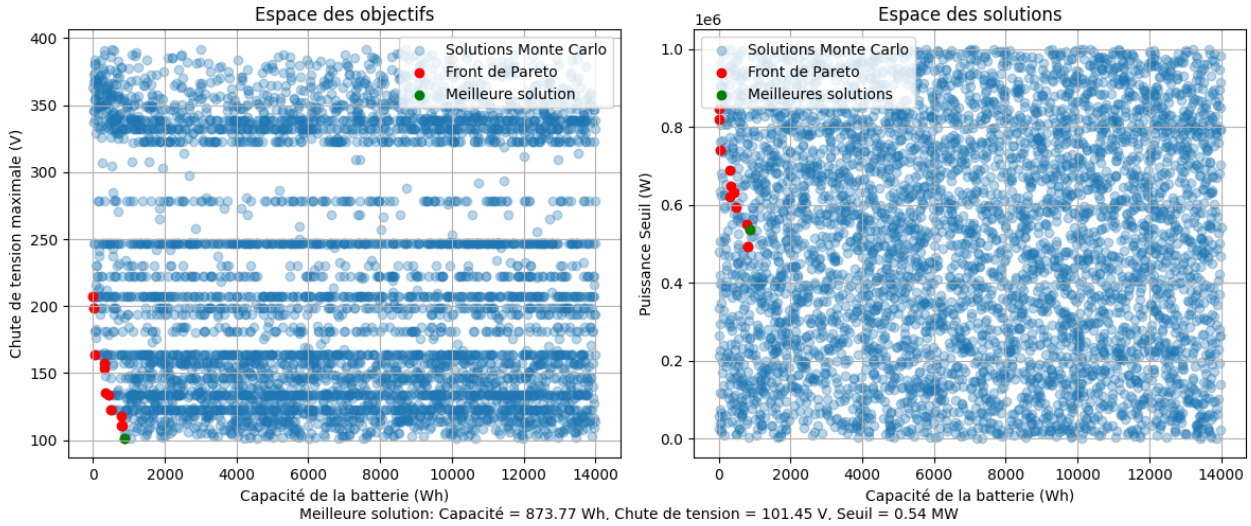

FIGURE 17 : Espace des objectifs avec solutions non dominées et meilleure solution pour $N = 5000$.

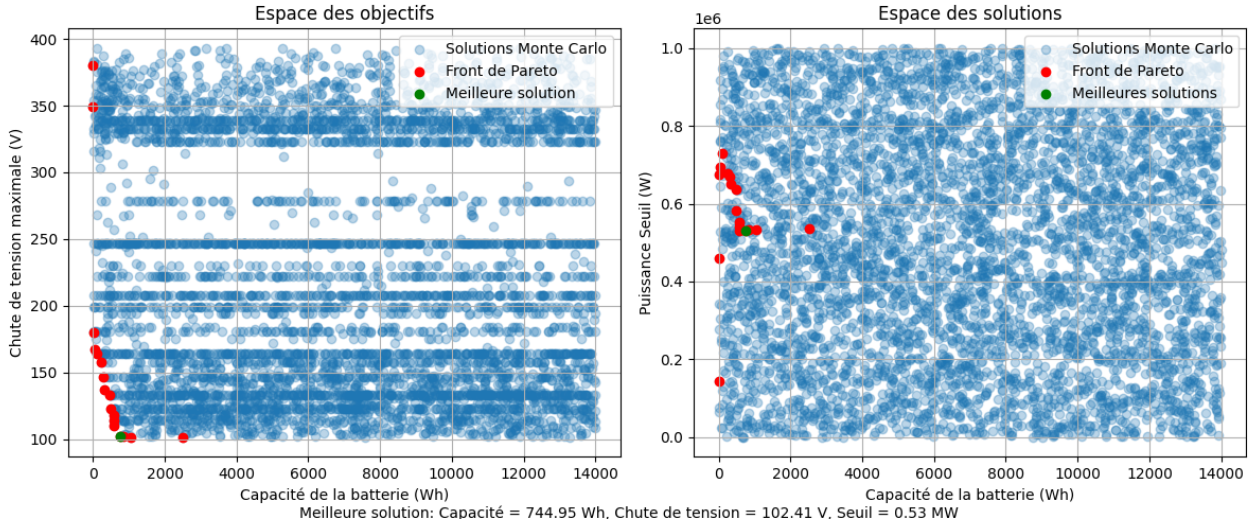

FIGURE 18 : Espace des objectifs avec solutions non dominées et meilleure solution pour $N = 5000$.

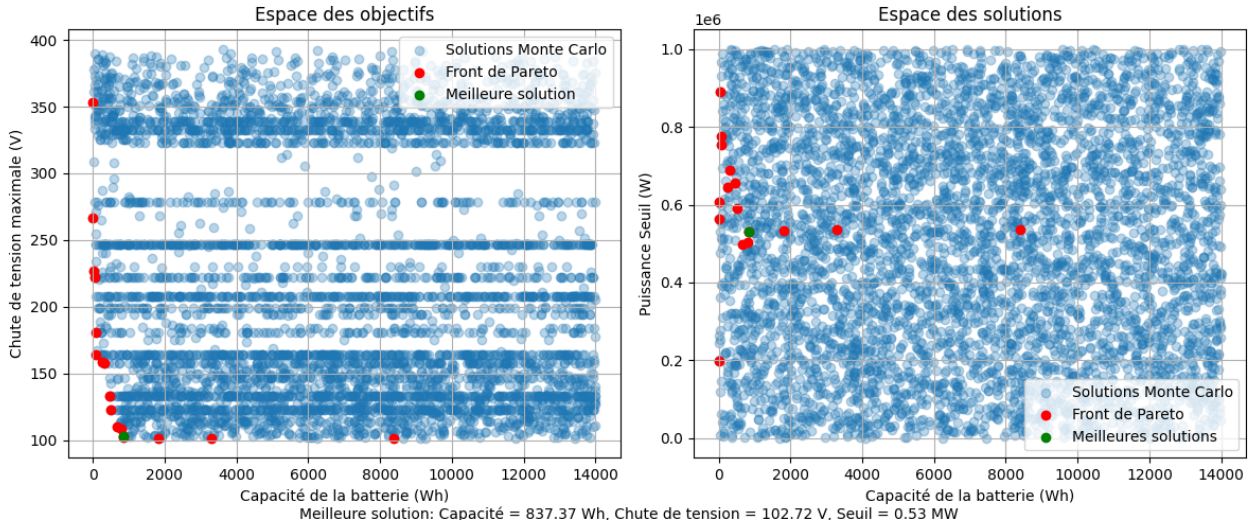

FIGURE 19 : Espace des objectifs avec solutions non dominées et meilleure solution pour $N = 5000$.

Conclusion Monte-Carlo

Nous constatons que notre chute de tension maximale tourne autour de 110 V pour $N = 1000$ couples et autour de 103 V pour $N = 5000$ couples. En augmentant le nombre d'échantillons, nous obtenons en effet un résultat plus optimale.

Algorithme NSGA-II

L'algorithme génétique NSGA-II (*Non-Dominated Sorting Generic Algorithm 2*) permet de trouver les meilleures solutions dans un contexte donné de façon assez efficace. Cet algorithme génère une population de taille N d'individus (les solutions), puis recherche au sein de cette population les « solutions non-dominées », et les range en suivant cette ordre :

— En premier, les solutions non-dominées, c'est-à-dire celles qui dominent toutes le autres, les meilleures de la population.

— En deuxième, les solutions non-dominées qui restent, en ayant retirées les solutions précédentes de la population de départ.

— En troisième, les solutions non-dominées restantes, en ayant retirées toutes les précédentes de la population de départ.

— On continue jusqu'à épuisement des individus dans la population de départ.

Ayant maintenant à notre diposition un classement des individus selon leurs performances, on sélectionne les 50% meilleurs dans ce classement, puis les croisons afin de créer la génération suivante. Puis le cycle recommence, pour un total de N itérations.

Il faut savoir que NSGA-II, demandant un classement des solutions selon leur dominance, doit utiliser une fonction qui classe les solutions selon cette ordre. Pour ceux qui regarderaient notre code, il s'agit de la fonction `non_dominant_sort()` (voir figure 20).

```python
def non_dominant_sort(pop):
    """
        Renvoie les rangs, contenant différents individus suivant la dominance de ces derniers (1er rang = les non-dominées, 2e rang = les autres non-dominés, etc).
    """
    fronts = []
    front = [] # Premier rang.
    population = pop[:] # Initialisation de la population.
    while len(population) > 0:
        for individual in population:
            dominated = False # On suppose que l'individu n'est pas dominé.
            for other in population:
                if (other[0] <= individual[0] and other[2] <= individual[2] and other != individual):
                    dominated = True # L'individu est dominé.
                    break
            if not dominated:
                front.append(individual[:])
        if len(front) == 0:
            front = population[:]
        fronts.append(front[:])
        for individual in front:
            population.remove(individual) # On retire les individus déjà classés.
        front = [] # On réinitialise le premier rang.
    return fronts
```

FIGURE 20 : Code de la fonction du classement en fronts.

Et, pour sélectionner les 50% meilleurs, nous utilisons la fonction ci-dessous affichée en figure 21 :

```python
def select_half_best(fronts, popSize):
    """
        Retourne une liste des 50% meilleurs individus de rangs.
    """
    best = []
    for front in fronts:
        for individual in front:
            best.append(individual)
            if len(best) >= popSize // 2:
                return best
    return best
```

FIGURE 21 : Sélection des 50% meilleurs.

Comme affiché sur la figure 22, les solutions intéressantes pour notre batterie semblent se trouver aux alentours de 1,5 ou 2 kWh, pour un seuil vers les 300 kW et une chute de 200 V (les points rouges représentant la dernière génération créée par notre algorithme). Suivant la simulation, les résultats peuvent énormément changer, et il est conseiller de relancer plusieurs fois l'algorithme afin de trouver des solutions potentiellement meilleures. Par ailleurs, étant donné l'absence de « *crowding_distance* », nommé « critère de distance » en français, nos solutions tendent à se rassembler en un amas compact ; augmenter le taux de mutation peut sensiblement améliorer ce problème.

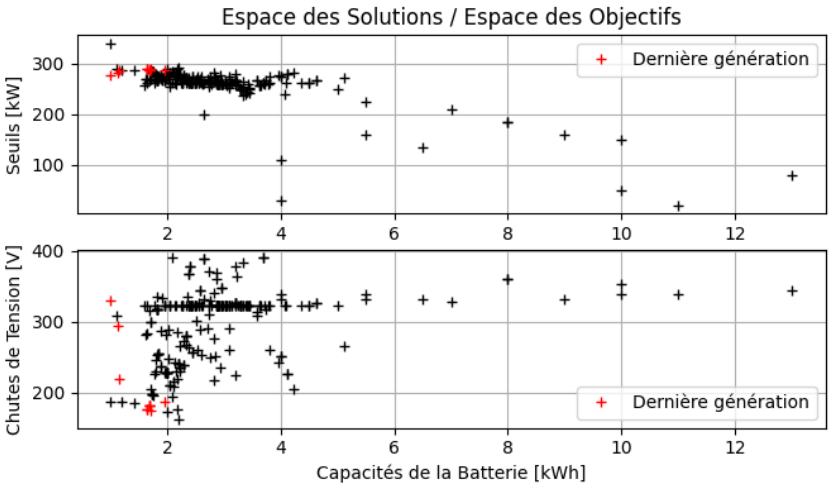

FIGURE 22 : Résultats de l'algorithme NSGA-II.

Conclusion

Pour conclure, nous pouvons dire que l'algorithme NSGA-II est plus efficace que la méthode Monte-Carlo ; en revanche, Monte-Carlo s'étend dès le départ sur un maximum de résultats, là où NSGA-II (en particulier notre version simplfifée) peut rester coincé à quelques valeurs seulement (d'où l'intérêt d'avoir une population suffisamment large, et un taux de mutation pas trop bas, mais trop haut non plus).

Comme résultat final, nous pouvons mettre en avant une batterie de capacité $1,5$ kWh et un seuil de 300 kW auquel le train commence à demander de l'énergie à la batterie. Les résultats sont donnés en figure 23, pour de tels paramètres.

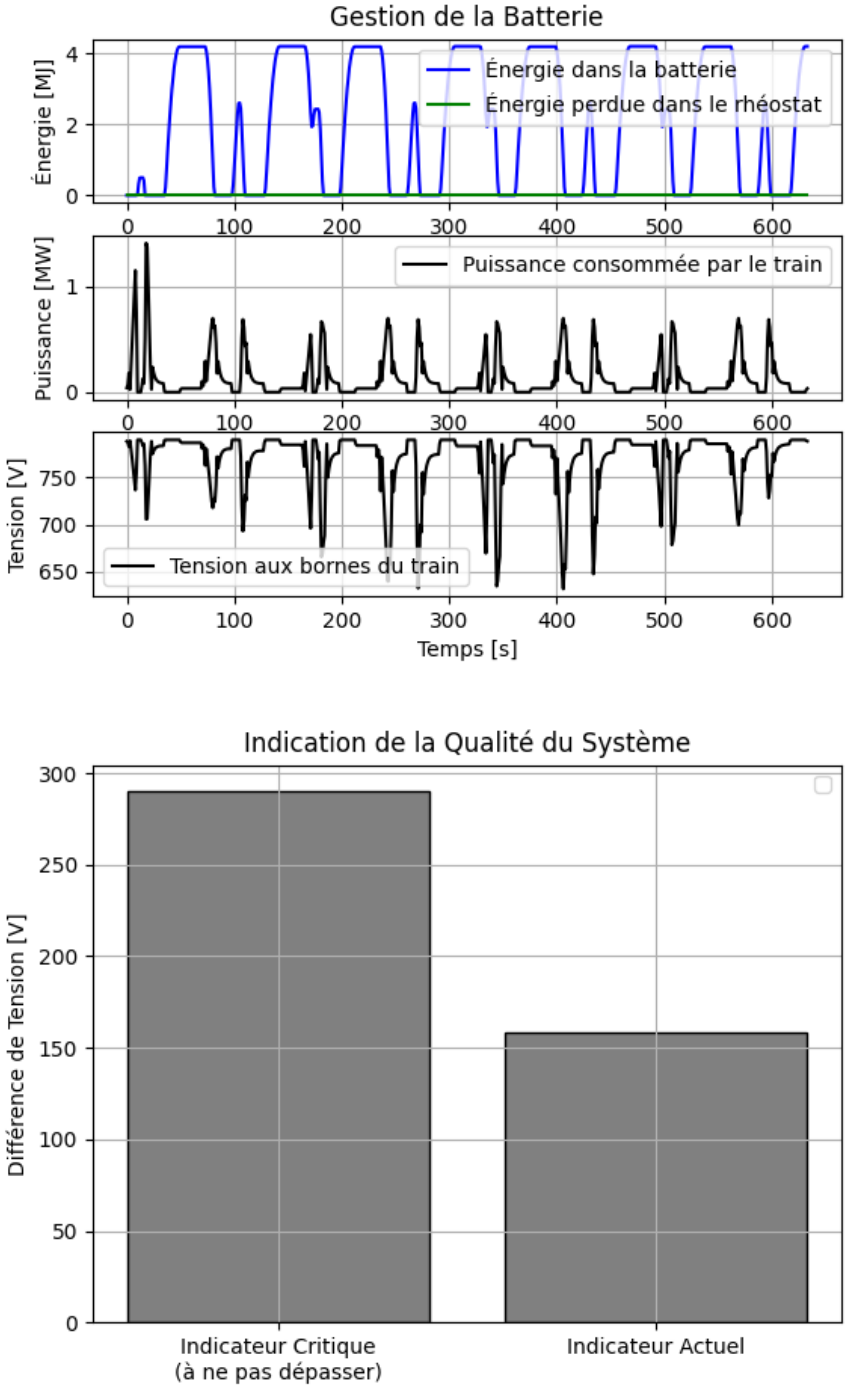

FIGURE 23 : Résultats de la simulation du train, pour une batterie de capacité $1,5$ kWh et un seuil de 300 kW (paramètres optimisés selon l'algorithme NSGA-II). L'indicateur de qualité est représenté par la chute de tension maximale aux bornes du train ; une chute de plus de 290 V est dangereuse.

Évidemment, d'autres batteries avec seuil optimisés sont également disponibles (revoir figure 22).