MU4MEN01 – Projet d'Optimisation

AKIL Adam, BUCLET Zeca, NOCHÉ Kévin

10 janvier 2025

Table des matières

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.

La fonction `Simulation()`

Sans Batterie

Avec Batterie

Monte-Carlo

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 regarderait notre code, il s'agit de la fonction `non_dominant_sort()` (voir figure 2).

```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 1 : Code de la fonction du classement en fronts.

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

```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 2 : Sélection des 50% meilleurs.

Comme affiché sur la figure 3, 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.

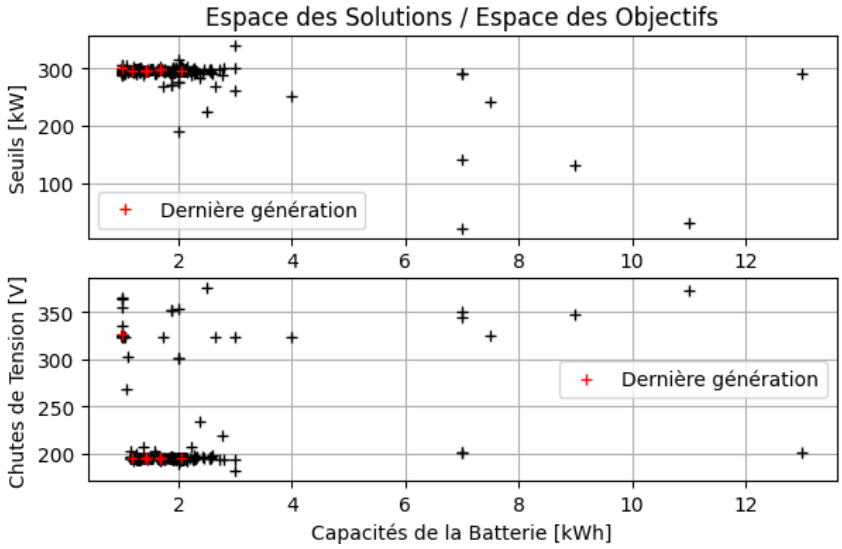

FIGURE 3 : Résultats de l'algorithme NSGA-II.

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 4, pour de tels paramètres.

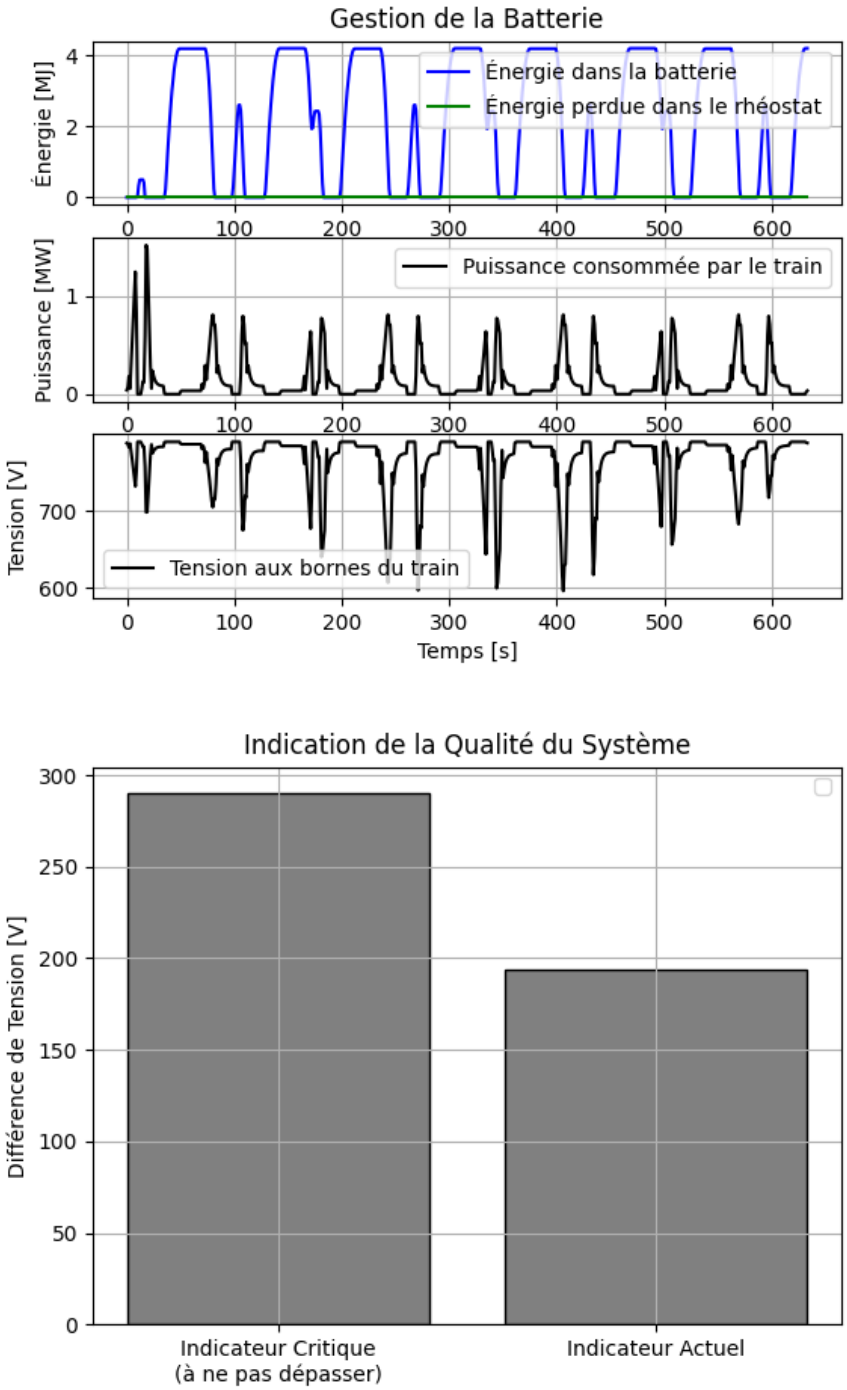

FIGURE 4 : Résultats de la simulation du train, pour une batterie de capacité 1,5 kWh et un seuil de 300 kW (paramètres optimisés selon l'algorithme NSGA-II).

Évidemment, d'autres batteries avec seuil optimisés sont également disponibles (revoir figure 3).