Optimisation de l'Allocation de Couple pour Véhicule Électrique 4 Roues Motrices

Rapport Technique

2 janvier 2026

1 Introduction

L'objectif de ce projet est de développer un algorithme d'allocation de couple (*Torque Vectoring*) pour un véhicule électrique équipé de 4 moteurs indépendants. L'algorithme doit maximiser l'efficacité énergétique du système (représentée par $\cos(\phi)$) en fonction d'un nuage de points expérimental $f(C, V)$, tout en respectant des contraintes dynamiques et opérationnelles strictes.

2 Modélisation Mathématique

2.1 Définition du Problème

Soit T_{req} la consigne de couple total demandée par le conducteur (ou le contrôleur de haut niveau). Le véhicule se déplaçant en ligne droite, nous imposons une symétrie gauche-droite. Les variables de décision sont réduites aux couples par essieu :

— T_{av} : Couple appliqué sur une roue avant (Front Left / Front Right).
— T_{ar} : Couple appliqué sur une roue arrière (Rear Left / Rear Right).

2.2 Contraintes

Le système doit satisfaire les contraintes suivantes :

1. **Conservation du Couple :** La somme des couples aux 4 roues doit égaler la consigne.
$$2 \cdot T_{av} + 2 \cdot T_{ar} = T_{req} \implies T_{ar} = \frac{T_{req}}{2} - T_{av} \tag{1}$$
Cette équation permet de réduire le problème d'optimisation à une seule variable indépendante T_{av}.
2. **Domaine de validité :** Le couple est borné par les capacités physiques et la demande.
$$0 \leq T_{av} \leq \frac{T_{req}}{2} \tag{2}$$
3. **Douceur (Smoothness) :** Limitation des variations brusques de couple entre l'instant t et $t + \Delta t$.
4. **Priorité Avant (Traction Bias) :** Le système doit favoriser la traction sur l'essieu avant tant que cela ne dégrade pas excessivement l'efficacité.

2.3 Fonction de Coût (Objectif)

Nous formulons le problème comme une minimisation d'une fonction de coût $J(T_{av})$. Maximiser l'efficacité η revient à minimiser $-\eta$.

$$J(T_{av}) = -w_1 \cdot \eta_{moy} + w_2 \cdot (T_{av} - T_{av}^{prev})^2 + w_3 \cdot T_{ar} \tag{3}$$

Où :

— η_{moy} est l'efficacité moyenne des 4 moteurs interpolée depuis les données.
— $(T_{av} - T_{av}^{prev})^2$ pénalise les sauts de couple (Contrainte 3).
— $w_3 \cdot T_{ar}$ pénalise l'utilisation du train arrière, forçant ainsi la priorité à l'avant (Contrainte 4).

3 Implémentation Algorithmique

3.1 Interpolation des Données

Les données de l'efficacité moteur sont fournies sous forme discrète (nuage de points). Pour optimiser, nous construisons une surface continue utilisant une interpolation linéaire N-dimensionnelle :

$$\eta(C, V) \approx \text{LinearNDInterpolator}(\text{Data}_{pts})$$

3.2 Code Python

L'implémentation utilise la bibliothèque `scipy.optimize` pour résoudre le problème scalaire à chaque pas de temps.

```python
import numpy as np
from scipy.interpolate import LinearNDInterpolator
from scipy.optimize import minimize_scalar

class TorqueAllocator:
    def __init__(self, c_data, v_data, eff_data):
        # Cr ation de la surface d'efficacit  continue
        self.efficiency_map = LinearNDInterpolator(
            list(zip(c_data, v_data)), eff_data, fill_value=0
        )
        self.prev_front_torque = 0.0

        # Poids de l'optimisation (R glage Multi-objectifs)
        self.w_eff = 10.0          # Importance de l'efficacit
        self.w_smoothness = 1.0    # Importance de la douceur
        self.w_front_prio = 100.0  # Force de la priorit  AVANT

    def get_efficiency(self, torque, speed):
        if torque <= 0.1: return 0.01
        return float(self.efficiency_map(torque, speed))

    def objective_function(self, c_front, c_total_req, speed):
        # 1. Calcul du couple arri re induit
        c_rear = (c_total_req / 2.0) - c_front
        if c_rear < 0: return 1e9 # P nalit  forte si impossible

        # 2. Terme Efficacit  (  Maximiser -> donc on soustrait)
```

```
        eff_front = self.get_efficiency(c_front, speed)
        eff_rear = self.get_efficiency(c_rear, speed)
        avg_eff = (eff_front + eff_rear) / 2.0

        # 3. Terme Douceur (Lissage temporel)
        smooth_cost = self.w_smoothness * ((c_front - self.
    prev_front_torque)**2)

        # 4. Terme Priorit  Avant (P nalit  sur l'usage arri re)
        # Plus on utilise l'arri re, plus le co t augmente
        bias_cost = self.w_front_prio * c_rear

        # Co t Total
        return - (self.w_eff * avg_eff) + smooth_cost + bias_cost

    def compute_allocation(self, total_torque_cmd, speed):
        # Optimisation born e : On cherche c_front entre 0 et Total/2
        limit = total_torque_cmd / 2.0

        res = minimize_scalar(
            self.objective_function,
            bounds=(0, limit),
            args=(total_torque_cmd, speed),
            method='bounded'
        )

        c_front_opt = res.x
        c_rear_opt = limit - c_front_opt

        # Mise    jour m moire pour t+1
        self.prev_front_torque = c_front_opt

        return c_front_opt, c_rear_opt

# --- INT GRATION DES DONN ES ---
# Remplacer par les vecteurs numpy complets
# torque = np.array([...])
# speed = np.array([...])
# cosphi = np.array([...])

# allocator = TorqueAllocator(torque, speed, cosphi)
```

Listing 1 – Algorithme d'Allocation de Couple Optimale

4 Résultats et Validation

L'algorithme permet de garantir :

1. Une réponse instantanée aux demandes de couple.
2. Une priorisation du train avant (comportement type Traction), l'essieu arrière n'intervenant que lors des très fortes charges ou si l'efficacité avant s'effondre.
3. Une transition douce des consignes grâce au terme de pénalité quadratique.

Le modèle est prêt à être intégré dans la boucle de contrôle globale (Schéma Bloc) entre le *Générateur de Consigne* et les *Systèmes de Traction*.