Grenoble INP - Projet 3A SEOC

Projet Systèmes embarqués et objets connectés

16 Janvier 2022

Contrôle-commande temps réel des systèmes Cyber-Physiques

Groupe : 08

Membres du groupe :

MESTRE Yannis

HASSY Hafssa

Encadrants :

MANCINI Stéphane

Martinez-Molina John-Jairo

Table des matières

Introduction

Ce rapport concerne le projet de contrôle en temps réel d'une petite voiture électrique sur une piste. Le projet est une mise en application des méthodes vues pendant le cours "Contrôle-commande temps réel des systèmes cyber-physiques".

Le contexte du projet est le suivant : la voiture dispose de deux roues arrière motorisées, dont les tensions d'alimentation peuvent être contrôlées indépendamment, et de deux roues avant libres, mais dont l'orientation peut être contrôlée via un servomoteur. De plus, la vitesse des roues arrière peut être mesurée grâce à 9 aimants disposés sur chaque roue et une caméra monochrome d'une résolution de 128x1 pixels permet de voir ce qui se trouve au sol devant la voiture. Un microcontrôleur permet de commander ces actionneurs et de lire ces capteurs, à l'aide de bibliothèques logicielles permettant de s'y interfacer facilement.

L'objectif est alors d'implémenter des lois de commande permettant à la voiture de d'avancer tout en restant sur une piste blanche dont les bords sont délimités par des bandes noires.

Pour y parvenir, nous avons procédé par étapes en commençant par contrôler la vitesse des moteurs, puis en contrôlant la direction et en adaptant les vitesses des moteurs à la courbe prise par la voiture. Enfin nous avons utilisé la caméra pour détecter le centre et les bords de la piste et piloté la direction et la vitesse selon cette mesure. Ces étapes seront présentées dans les sections qui suivent, après un aperçu de l'architecture logicielle du projet.

I - Architecture générale

Le code qui a été écrit pour ce projet s'appuie sur les bibliothèques bas niveau d'accès au matériel et sur FreeRTOS. Il consiste en un certain nombre de tâches, ordonnancées avec des priorités fixes. En particulier, la tâche principale du programme, qui s'exécute toutes les 50 ms grâce à la fonction *vTaskDelayUntil* de FreeRTOS, s'occupe de déclencher la lecture des capteurs (vitesse des roues, caméra) puis de contrôler les actionneurs (moteurs, direction). L'ordre de ces actions est bien déterminé, puisqu'elles ne sont pas réalisées par d'autres tâches indépendantes, mais simplement par des fonctions appelées par la tâche principale. C'est ce que l'on peut voir dans le très court code de la tâche principale ci-dessous :

```
void task_Trajectory(void *pvParameters)
{
    TickType_t previousWake = xTaskGetTickCount();

    for(;;)
    {
        // 1- Capteurs :
        read_camera();
        if(cptTrajectory % 4 == 0)
            read_speed();

        // 2- Commandes (l'ordre a une importance !):
        update_steering(-0.025*(float)erreurDirection);

        if(cptTrajectory % 4 == 0)
        {
            if(sortiePiste == false)
            {
                update_motor(1.0);
            }
            else
            {
                update_motor(0.0);
            }
        }

        cptTrajectory++;
        vTaskDelayUntil(&previousWake, Ms(50));
    }
}
```

Code de la tâche principale

Les acquisitions et commandes liées à la caméra et à la direction ont lieu toutes les 50 ms, c'est-à-dire à chaque exécution de la tâche principale. En revanche, celles concernant la vitesse des roues ou le contrôle des moteurs n'ont lieu que toutes les 200 ms, soit une fois toutes les quatre exécutions de la tâche principale. C'est la raison de la présence des *if* avec un compteur et un modulo dans le code : ils servent "d'ordonnancement manuel". Enfin,

notons que la tâche principale ne constitue pas à elle seule la totalité du programme. Il existe notamment une tâche dédiée à l'acquisition des images par la caméra avec un temps d'exposition constant ; cette dernière sera mentionnée plus loin dans le rapport.

II - Contrôle des moteurs

A - En boucle ouverte

Dans un premier temps, notre but est de faire rouler la voiture sur la piste en mémorisant le chemin à suivre. Pour ce faire, nous passons par plusieurs étapes. Premièrement, nous utilisons les moteurs des roues arrière de manière à avancer en ligne droite sur 1 m.

Chaque roue est commandée par son propre signal PWM et a un sens de rotation, mais on donne à la fonction *Motor_UpdatePwm* les mêmes valeurs du signal et de la direction pour les deux moteurs, pour sa cohérence et un bon fonctionnement.

Pour aller en ligne droite et s'arrêter, on utilise la fonction *vTaskDelay(2000)* qu'on ajoute entre le démarrage des moteurs et leur arrêt.

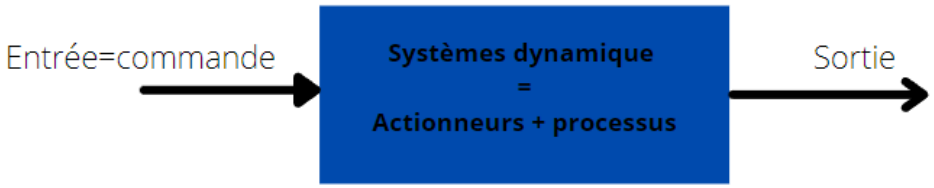
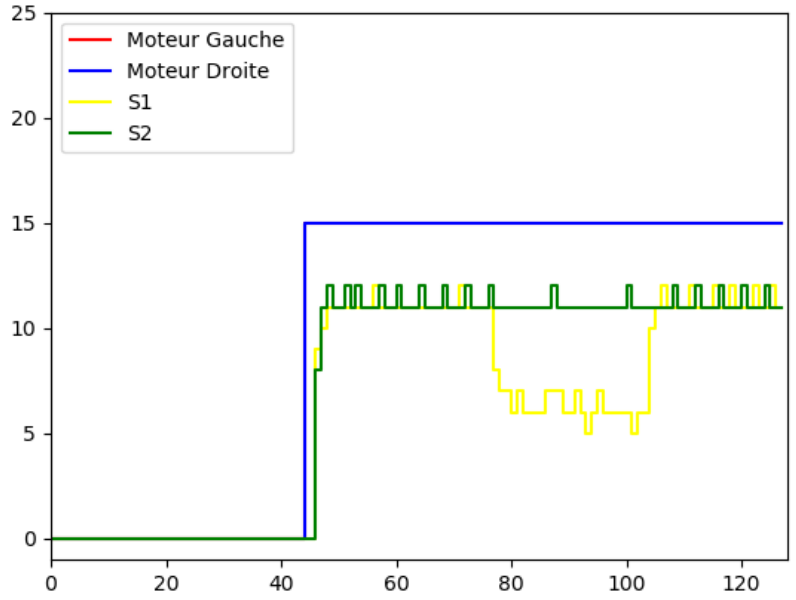


Schéma du système pour un moteur commandé en boucle ouverte

On remarque que la voiture ne roule pas en ligne droite à cause de la différence des deux vitesses des deux roues. Aussi, il n'est pas possible de s'assurer que la voiture parcourt bien 1 mètre si on ne mesure pas sa vitesse. Le délai de 2 secondes ajouté entre le démarrage et l'arrêt des moteurs est approximatif, empirique et valable seulement pour la voiture utilisée.

Enfin, la boucle ouverte ne permet pas de maîtriser la vitesse des roues de manière robuste, c'est-à-dire de rester à une vitesse constante quand les conditions (par exemple de frottements) changent. La figure suivante le montre : elle représente la vitesse de rotation des roues arrière en fonction du temps, pour une tension d'alimentation constante, et avec un frottement ajouté sur l'une des deux roues.



Vitesse de rotation des roues motrices (courbes jaune et verte) en fonction du temps. Les courbes rouge et bleu (confondues) correspondent à la tension d'alimentation des moteurs.
Un peu avant t=80 on touche une des roues pour la ralentir et on la relâche après t=100. On voit que le système, qui ne mesure pas la vitesse des roues, ne s'est pas adapté à ce changement de conditions.

C'est pour cela qu'on a besoin d'asservir notre système. Cet asservissement peut être réalisé soit par une commande intégrale simple, soit par une modélisation du système plus complexe.

B - En boucle fermée

Pour réaliser ce contrôle, il faut nécessairement calculer la vitesse de chaque roue. On se servira du compte tour, réalisé à l'aide d'un capteur à effet Hall et d'aimants placés sur la roue. Chaque passage d'un des 9 aimants d'une roue devant le capteur incrémente un compteur, qui est relevé toutes les 200 ms. On peut donc changer l'unité de la vitesse calculée de 1/9 tour /200ms à m/s. Ayant mesuré le diamètre des roues (6,3 cm), nous pouvons faire cette conversion en multipliant la valeur relevée par 0,1099 (1/9 tour /200ms= 1/9*6.3*Π /0.2/100=0.1099m/s).

A l'aide de ce même compte tour, on définit dans la tâche de **Task_Speed.c** deux variables (poisition1, position2) contenant la distance parcourue par les roues que nous utilisons dans **Task_Trajectory.c** pour savoir si la voiture a parcouru 1 m et envoyer la consigne de vitesse correspondante. La distance parcourue par la voiture est la moyenne des deux valeurs.

1 - Avec une commande intégrale simple

Revenant à notre asservissement, on applique une commande “intégrale” de la forme:

$$PWM(k+1) = PWM(k) - K(\omega - \omega\_ref)$$

Avec w la vitesse angulaire mesurée, w_ref la vitesse angulaire cible.

Dans notre code, cette loi de commande est présenté sous la forme suivante :

```
pwm1 = pwm1 - K * (s1n - consigneVitesse1);
pwm2 = pwm2 - K * (s2n - consigneVitesse2);
```

Avec s1n et s2n les vitesses mesurées.

On essaie différentes valeurs pour K et on remarque que la correction est soit lente soit brusque. On trouve que pour K=0.5 (pour des vitesses exprimées en 9ème de tour / 200 ms) l’asservissement est un peu meilleur, mais pas totalement satisfaisant non plus.

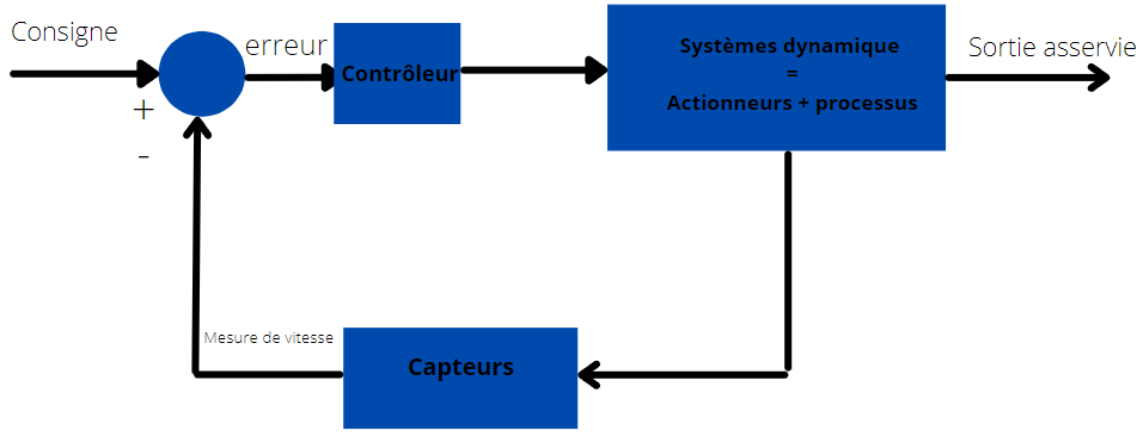


Schéma du système pour un moteur commandé en boucle fermée (avec mesure de sa vitesse de rotation).

2 - Modèle en représentation d'état :

Pour avoir une commande plus efficace des moteurs, on modélise chaque moteur avec un vecteur de représentation d’état. Le but est de prendre en compte dans la correction à la fois l’écart à la consigne immédiat et son intégrale.

On considère pour un seul moteur notre vecteur d’état étendu X et le vecteur de commande U :

$$X = \begin{vmatrix} W \\ Z \end{vmatrix}, U = V$$

avec w: la vitesse angulaire de la voiture , Z(k+1) = Z(k) + (r-w) intégrale de l'erreur de suivi où r est la consigne de vitesse et V la tension appliquée aux moteurs.

On utilise les équations physiques simplifiant le fonctionnement de la voiture :

$$J\dot{\omega}_d = \frac{K_e}{R_m}(V_d - K_e\omega_d)$$
$$J\dot{\omega}_g = \frac{K_e}{R_m}(V_g - K_e\omega_g)$$

Avec :

- J : le moment d'inertie pour un moteur
- K_e: le coefficient de force électromotrice des moteurs
- R_m: la résistance électrique des moteurs
- V_g et V_d: les tensions d'alimentations des moteurs gauche et droit
- ω_g et ω_d: les vitesses de rotation des roues gauche et droite

Les valeurs utilisées pour tous ces paramètres sont issues du même document qui contient le modèle de la voiture. Nous savons qu'elles ne viennent pas de mesures précises, mais font tout de même l'affaire, comme on va le voir par la suite.

On trouve notre modèle de la forme :

$$\dot{X} = AX + BU$$

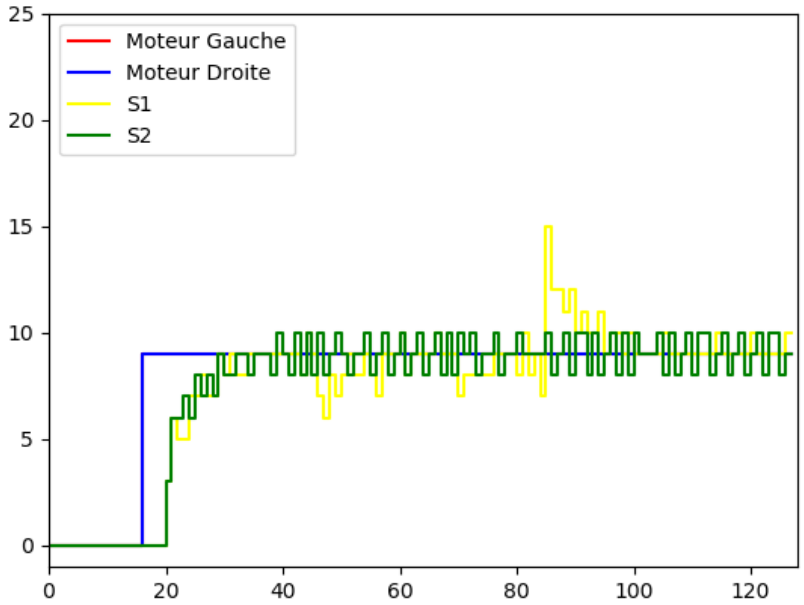
Dans un code python sur PC, on calcule la matrice de contrôle optimale K par dlqr (discrete linear quadratic regulator) en utilisant les valeurs des matrices Ad et Bd en temps discret(Ad=(I+Te*A), Bd=Te*Bd, ici Te = 200 ms, la période de la boucle de contrôle des moteurs) et deux matrices de pondération. On aura donc comme commande pour chaque moteur U = -K X.

Dans le code C, cette loi de commande est implémentée comme il suit :

```
z1 = z1 + ((float)s1n - consigneVitesse1);
z2 = z2 + ((float)s2n - consigneVitesse2);

// Coefficients issus d'un calcul par DLQR :
float pwm1 = (-0.39894094 * ((float)s1n-consigneVitesse1) - 0.23036774 * z1);
float pwm2 = (-0.39894094 * ((float)s2n-consigneVitesse2) - 0.23036774 * z2);
```

En implémentant cette méthode d'asservissement sur notre système et en appliquant des frottements sur la roue gauche, on remarque que le système s'adapte rapidement à ce changement. La figure suivante démontre cette expérimentation.



Vitesse de rotation des roues motrices (courbes jaune et verte) en fonction du temps. Les courbes rouge et bleu (confondues) correspondent à la consigne de vitesse. Vers t=50 on touche une des roues pour la ralentir et on la relâche après t=80. On voit que le système s'adapte rapidement à ce nouveau frottement et conserve une vitesse très proche de la consigne. On remarque aussi que la vitesse du moteur dépasse la consigne au moment où on relâche la roue, mais retrouve ensuite la consigne.

III - Contrôle de la direction

A - À l'aveugle

Pour que la voiture puisse suivre la piste sans la voir, il faut obligatoirement encoder la forme de la piste dans le programme qui contrôle la voiture, d'une manière ou d'une autre. Nous l'avons fait sous la forme d'un automate à états dont chaque état correspond à une portion de la piste, selon le schéma ci-dessous.

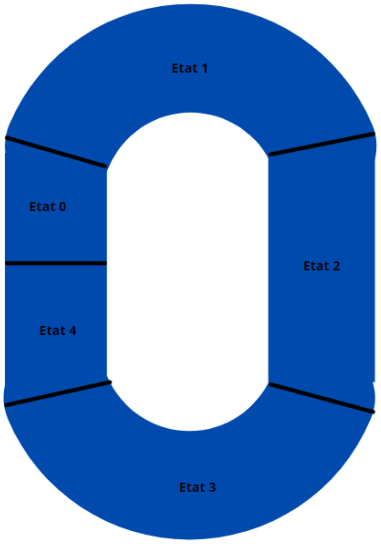


Schéma de la piste de test.
Chaque portion délimitée correspond à une vitesse et à une direction pour la voiture.

Un état donné fixe une vitesse de consigne pour les moteurs (on choisit une allure plus faible dans les courbes que dans les lignes droites) et un angle pour les roues avant. L'implémentation de cet automate s'est faite simplement avec une variable stockant l'état et une boucle *for* contenant un *switch* dont chaque *case* correspond à un état. Les transitions d'états se faisaient en fonction de la distance totale parcourue par la voiture depuis son démarrage.

Après quelques ajustements par essais successifs de l'angle des roues avant dans les virages, cette solution s'est avérée suffisamment précise pour que la voiture fasse deux tours complets de la piste sans en sortir. Bien sûr, ce n'est pas une solution viable pour tourner indéfiniment puisque le moindre écart est amplifié à chaque nouveau tour de piste, n'étant jamais mesuré et donc jamais corrigé.

B - Avec la caméra

1 - Acquisition d'images

Pour détecter un écart entre l'orientation de la voiture et la direction de la piste, on utilise une caméra qui produit une image en niveaux de gris de 128 pixels de large et un seul de haut. Lorsque l'on déclenche une prise de vue, on récupère une image dont le temps d'exposition est celui écoulé depuis la dernière prise de vue. Pour contrôler ce temps d'exposition, on doit donc déclencher deux prises de vues espacées d'une pause de la durée voulue et ne garder que l'image de la deuxième prise. C'est ce que nous faisons, dans une tâche indépendante dédiée. La "double intégration" prend alors son sens, puisque nous fixons le temps d'exposition à 6 ms, tandis que la période de la tâche est de 40 ms pour ne pas utiliser toute la capacité du processeur.

En créant simplement cette tâche comme n'importe quelle autre, nous avons d'abord obtenu des images très instables, avec des zones saturées qui changeaient complètement d'emplacement d'une prise de vue à l'autre. Ce problème était en fait dû à la priorité accordée à cette tâche qui était trop faible. Une autre tâche plus prioritaire interrompait souvent l'acquisition, ce qui amenait à un temps d'exposition différent sur deux parties d'une même image. Nous avons réglé ce problème en rendant la tâche de prise de vue la plus prioritaire du programme. Une représentation de ce que voit alors la caméra est donnée sur la figure ci-dessous.

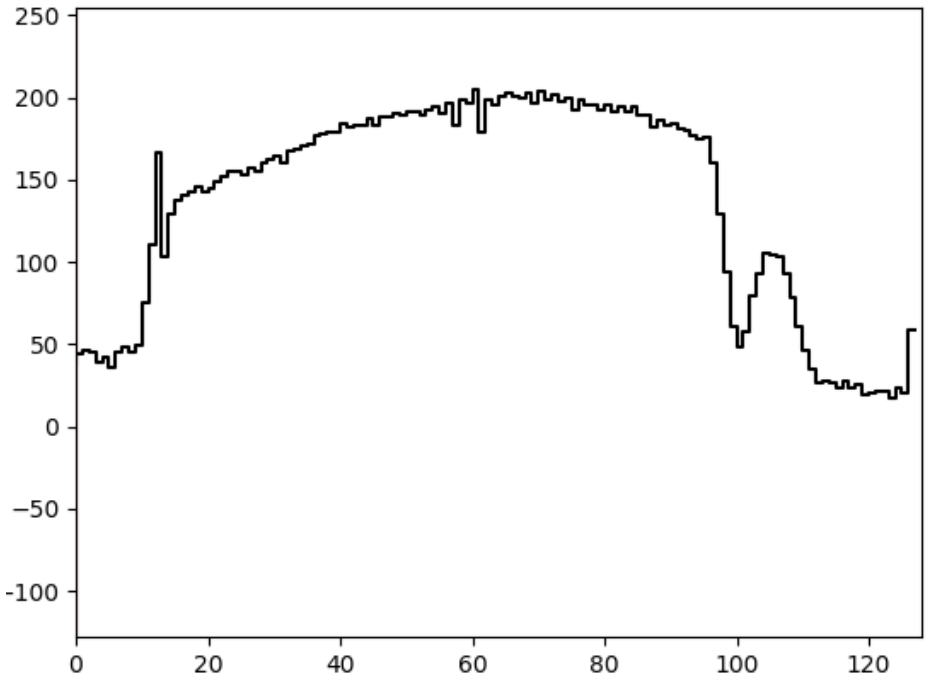


Image prise par la caméra : luminosité en fonction de l'abscisse, en pixels.
On voit le milieu de la piste, qui est blanc, les bords noirs et le le reste de l'environement, ici assez sombre.

Afin d'être plus robuste aux conditions d'éclairage, nous avons essayé d'adapter automatiquement le temps d'exposition à la lumière ambiante. Le principe était de prendre un temps d'exposition tel que la luminosité moyenne, calculée sur tous les pixels, soit toujours égale à une constante. Malheureusement, nous n'avons rien obtenu d'exploitable. Nous restons donc sur une durée de 6 ms, ce qui donne de bons résultats dans le cadre de ce projet, l'éclairage étant très stable.

2 - Traitement des images et décision

Une image est un tableau de pixels, ici une simple liste de 128 valeurs. Il faut alors en extraire une information utile : l'objectif est de déterminer où se trouve la piste par rapport à l'orientation de la voiture, autrement dit on cherche le milieu de la piste sur l'image. Pour cela on utilise un algorithme relativement simple qui parcourt une seule fois et entièrement l'image et trouve les bords gauche et droit de la piste. Pour qu'il fonctionne, on défini un seuil de luminosité qui sépare les pixels en seulement deux catégories : lumineux ou sombre, et on fait l'hypothèse que le milieu de la piste, qui est blanc, sera toujours détecté comme lumineux, et que les bords, noirs, seront détectés sombres. Cet algorithme est visible par son implémentation dans le fichier **Task_camera.c**.

Puisqu'on connaît les bords de la piste, on peut calculer l'indice du pixel qui se trouve en son milieu. Quand la voiture est parfaitement alignée avec la piste, le milieu de la piste doit coïncider avec le milieu de l'image. Si le milieu de la piste est à droite de l'image, c'est qu'il faut tourner à droite, de même à gauche. On peut donc calculer une erreur de direction, qui est la distance (en pixels) entre le milieu de l'image et le milieu de la piste.

Lorsque la voiture roule, on voudrait qu'à tout instant l'orientation des roues avant suive la piste, c'est-à-dire que les roues avant soient parallèles aux bords de la piste, ou aux bords de la piste un peu plus loin devant la voiture. Comme le servomoteur relié à la direction dispose de son propre asservissement, on se contente de régler sa consigne à chaque fois exactement de manière à aligner les roues avant avec la piste. Pour les angles concernés, assez faibles, on considère que la conversion de l'erreur de direction en pixels vers un angle est linéaire. Le facteur de proportionnalité, complètement déterminé par l'expérience, est alors de 0,025 rad/px.

Conclusion

Pour conclure, l'avancement final du projet est le suivant : la voiture peut suivre la piste sans problème, en étant à sa vitesse de consigne de 1 m/s. Cependant, notre solution est peu robuste à des changements d'éclairage ou de couleur et propreté de la piste, étant donné que le temps d'exposition de la caméra est réglé à une valeur fixe. C'est une limite qui serait certainement rédhibitoire pour participer à la *NXP cup*, mais qui n'est pas gênante dans le cadre du projet, l'éclairage de la salle étant principalement dû aux lampes au plafond en ces temps hivernaux.

La réalisation de ce projet a en tout cas été l'occasion de confronter nos bases théoriques vues en cours à la pratique, et de réaliser que ce n'est pas toujours simple. Comme vu dans la partie II-B, nous avons tout de même pu appliquer efficacement le résultat d'une modélisation en représentation d'états d'un moteur de la voiture.