UNIVERSITÉ TOULOUSE III PAUL SABATIER

Université de Toulouse

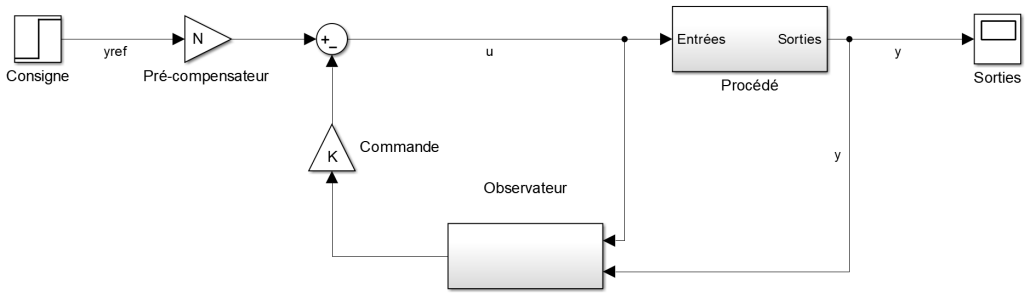
UNIVERSITÉ PAUL SABATIER

Conception et mise en œuvre de commande à temps réel

- Rapport final -

Auteurs :
Lucien RAKOTOMALALA
David TOCAVEN

Encadrants :
Sylvain DUROLA
Frédéric GOUAISBAUT
Yann LABIT



Faculté des Sciences et d'Ingénierie

EEA TOULOUSE

28 janvier 2018

Table des matières

Table des figures

Introduction

Pour l'UE *Conception et mise en œuvre de Commande à temps réel*, nous devons réaliser la commande d'un système temps réel, à partir de la modélisation du procédé jusqu'à l'implémentation sur un micro-contrôleur.

Notre problématique est la suivante : asservir un banc de moteurs à courant continu via un micro-contrôleur C167. Pour y parvenir, nous allons utiliser des références du Master EEA ISTR, 1er et 2ème année, ainsi que des ressources matérielles et logicielle disponibles dans les salles de TP de l'université. L'objectif de ce rapport est de vous présenter l'ensemble de nos travaux pour répondre à cette problématique. Ceux-ci ont été séparé en deux parties qui vous sont présentés ci dessous.

Première partie La première partie de ce rapport contient toute la partie théorique. Celle-ci est décomposée dans les trois premiers chapitres dont nous faisons la liste ci dessous.

Le *chapitre 1 : Modélisation* contient l'étude physique qui nous a été donné en cours, le modèle le plus précis, non linéaire et variant, les différentes simplifications de celui-ci.

Ensuite, dans le *chapitre 2 : Analyse*, nous avons effectué une analyse de nos différents modèles afin de maîtriser l'impact de nos simplifications, étudier les performances de notre systèmes et définir celles souhaitées. Nous avons réalisé cela grâce, autant que nous avons pu, a une approche théorique et grâce à des simulations.

Le *chapitre 3 : Synthèse de commande*, qui est le dernier chapitre de la partie théorique, contient la conception de l'asservissement et l'étude des performances de celle-ci sur les différents modèles de notre système.

Seconde partie Pour cette seconde partie, nous allons nous intéresser à l'implémentation de la commande. Nous serons dans un cadre de conception et validation de prototype à l'aide de protocole.

Le *chapitre* **??** *Validation de la commande en temps continue sur le modèle non linéaire* va permettre d'introduire les protocoles utilisés pour valider les prototype de commande. Une attention sera porté pour chaque prototypes crées à partir des protocoles, pour connaître leur performances vis à vis du cahier des charges.

Dans le *chapitre* **??** *Commande à temps discret*, nous étudierons les moyens à notre disposition pour rendre la commande à temps discret, et nous discrétiserons les prototypes précédemment validé.

Enfin, le *chapitre* **??***, Implémentation* portera sur la programmation de la loi de commande sur le C167. Nous devons alors pour cela, relever les problématiques liés à l'électronique et l'informatique du micro-contrôleur, pour enfin expliquer la démarche et la mise en œuvre réalisées.

1 Modélisation

1.1 Choix du formalisme et de la modélisation

Notre modélisation sera basée sur les modèles physiques qui décrivent les différents constituants de notre système de procédé : deux moteurs couplés l'un à l'autre par un arbre simple. L'un étant générateur de force mécanique et l'autre générateur de courant afin de faire office de charge (il dissipe son énergie sur une résistance). Il y a aussi un tachymètre couplé à l'arbre principal par un réducteur. Nos modèles seront donc des modèles de connaissances.

Nous avons choisis de faire une modélisation espace d'état pour différentes raisons. La première est que cette représentation permet d'étudier facilement la valeur des différents états (l'étude de la stabilité asymptotique, par exemple, en est simplifiée dans un modèle espace d'état). Elle permet aussi de garder les états non observables et non commandables dans le modèle, une modélisation par fonctions de transferts ne le permet pas. Ce choix nous permet aussi, pour la suite, de concevoir un asservissement par retour d'état basé observateur, qui est l'asservissement que nous avons choisis. Le choix d'un modèle de connaissance améliore aussi l'analyse de l'influence des différents paramètres du modèle, ce qui nous permettra d'affiner notre modèle lors des tests sur le système réel.

1.2 Maquette et équations physiques

1.2.1 Maquette :

Voici, figure 1.1, un schéma électrique et physique du banc qui fait office de procédé dans notre asservissement. Le circuit électrique de gauche correspond au Moteur à Courant Continue 1 (MCC) qui délivre la puissance

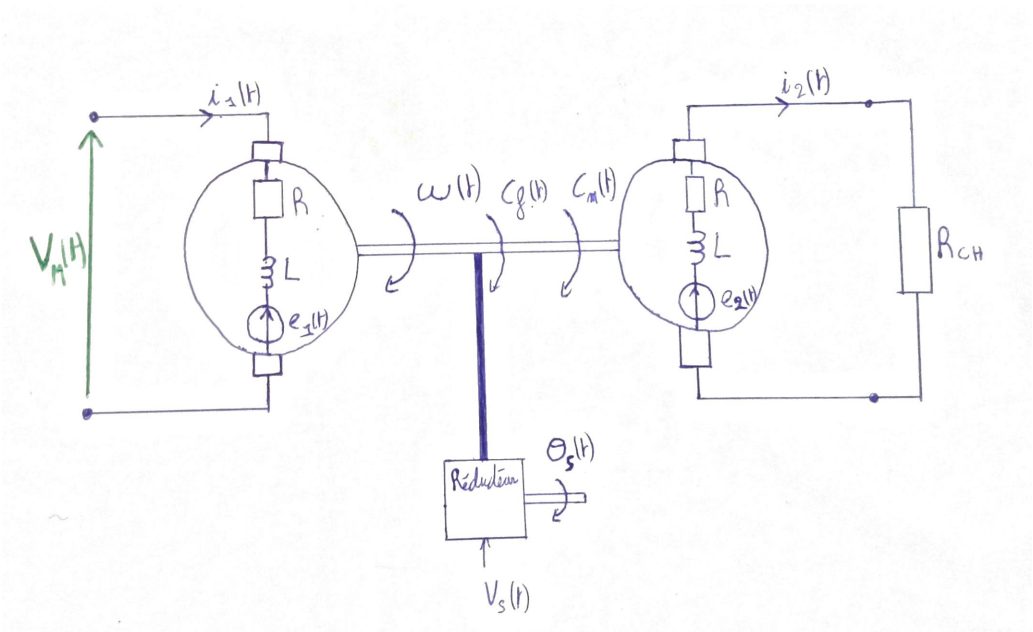


FIGURE 1.1 – Schéma électrique/physique du banc moteur

mécanique à partir d'une tension d'entrée V_M. Celle-ci est l'entrée de notre procédé. Le schéma électrique de droite représente le MCC 2, générateur de puissance électrique qui fait office de charge en alimentant une résistance R_{CH}. Entre ces deux schémas électriques, se trouve une représentation de l'arbre, des forces qu'il

subit, du réducteur et du tachymètre qui délivre une tension proportionnelle à la position du moteur, c'est le signal V_s, nous l'étudierons en tant que sortie.

1.2.2 Équations physique

Voici les différentes équations décrivant notre procédé :

- Équations des moteurs :

$$V_m(t) = Ri_1(t) + L\frac{di_1(t)}{dt} + e_1(t) \quad (1.1)$$

$$e_2(t) = (R + R_{CH})i_2(t) + L\frac{di_2(t)}{dt} \quad (1.2)$$

- Équations banc :

$$J_2\frac{d\omega(t)}{dt} = C_m(t) + C_f(t) \quad (1.3)$$

$$\frac{d\theta_s(t)}{dt} = \omega(t) \quad (1.4)$$

$$e_1(t) = K_e\omega(t) \quad (1.5)$$

$$e_2(t) = K_e\omega(t) \quad (1.6)$$

$$C_m(t) = K_c i_1(t) - K_c i_2(t) \quad (1.7)$$

$$C_f(t) = -\mu\omega(t) - C_0(t) \quad (1.8)$$

$$R_{CH}(t) = R_{CHn} + rR_{CH}(t)\Delta_1 \quad (1.9)$$

$$C_0(t) = -sign(C_m(t)C \quad (1.10)$$

$$V_s(t) = K_r K_s \theta_s(t) \quad (1.11)$$

$$V_g(t) = K_g\omega(t) \quad (1.12)$$

Où :

R La résistance de l'induit aux moteurs.

L L'inductance de l'induite des moteurs.

$i_1(t), i_2(t)$ Respectivement le courant dans les moteurs 1 et 2.

$e_1(t), e_2(t)$ Respectivement la force électromotrice des moteurs 1 et 2.

$R_{CH}(t)$ Résistance de charge.

J_2 Inertie totale (somme des inerties du rotor, du réducteur et de la charge).

$\omega_s(t)$ Vitesse radiale de l'arbre.

θ_s Position de rotation de l'arbre.

$C_m(t)$ Couple de l'arbre.

$C_f(t)$ Couple de frottement.

K_e Constante de force électromagnétique.

K_c Constante de couple.

μ Coefficient de frottement visqueux.

$C_0(t)$ Couple de frottement sec si rotation.

R_{CHn} Résistance de charge nominale.

rR_{CH} Rayon de l'incertitude de la résistance de charge.

Δ_1 Perturbation de la résistance de charge bornée en $[-1; 1]$.

K_r Facteur de réduction.

K_s Constante du potentiomètre.

K_g Constante de la génératrice tachymétrique.

$V_g(t)$ Tension reflétant la vitesse de rotation de l'arbre. Sortie de performance non mesurée.

C Couple de frottement sec si $\omega(t)$ assez grande.

1.2.3 Linéarisation

Les équations ci-dessous permettent de former un modèle du procédé. Néanmoins, l'équation 1.10 exprime une non linéarité sur les frottements secs du banc moteur. Afin d'avoir un modèle linéaire du moteur, nous avons transformé cette non linéarité en incertitude. Ainsi nous exprimons :

$$C_0(t) = C_{0n} + rC_0\Delta_2 \tag{1.13}$$

Où :

C_{0n} Couple de frottement sec nominal.

rR_{CH} Rayon d'incertitude du couple de frottement sec.

Δ_1 Perturbation du couple de frottement sec bornée en $[-1; 1]$.

Nous avons maintenant des équations linéaires mais deux équations présentent des incertitudes dues aux perturbations Δ_1 et Δ_2 respectivement liés à la température de la résistance de charge et à la non linéarité du couple de frottement sec.

1.2.4 Invariance

Nous souhaitons exprimer le modèle du procédé sous la forme d'un espace d'état linéaire et invariant, donc il faut rendre invariant nos équations. Pour cela, nous allons considérer que la résistance de charge $R_{CH}(t)$, exprimée dans l'équation 1.9 et $C_0(t)$, dans l'équation 1.13 valent leurs valeurs nominales.

$$R_{CH}(t) = R_{CHn} \tag{1.14}$$
$$C_0(t) = C_{0n} \tag{1.15}$$

Nous pouvons maintenant former un modèle linéaire et invariant.

1.3 Modèle de niveau 0, modèle EE0

Notre modèle présente 4 dynamiques, nous avons donc 4 états dans notre vecteur d'état.

Le vecteur d'état choisit est :

$$x(t) = \begin{bmatrix} i_1 \\ i_2 \\ \Omega_m \\ \Theta_m \end{bmatrix} \tag{1.16}$$

L'entrée $u(t)$ du modèle est $u(t) = V_m(t)$.

Les sorties sont $y(t) = \begin{bmatrix} V_g(t) \\ V_s(t) \end{bmatrix}$.

Le modèle d'ordre 4 vaut donc :

$$\begin{cases} \dot{x}(t) = \begin{pmatrix} -\frac{R}{L} & 0 & 0 & -\frac{K_e}{L} \\ 0 & -\frac{(R+R_{ch})}{L} & 0 & -\frac{K_e}{L} \\ 0 & 0 & 0 & 1 \\ \frac{K_c}{J_2} & \frac{K_c}{J_2} & 0 & \frac{\mu}{J_2} \end{pmatrix} x(t) + \begin{pmatrix} \frac{1}{L} \\ 0 \\ 0 \\ 0 \end{pmatrix} u(t) \\ y(t) = \begin{pmatrix} 0 & 0 & 0 & K_g \\ 0 & 0 & K_r K_s & 0 \end{pmatrix} x(t) \end{cases} \tag{1.17}$$

1.4 Espace d'état d'ordre 3, modèle EE1

Afin de facilité la création d'une commande, nous avons retiré l'état $i_2(t)$ de l'état de la modélisation. Il permettra une validation intermédiaire.

L'état vaut maintenant

$$x(t) = \begin{bmatrix} i_1 \\ \Omega_m \\ \Theta_m \end{bmatrix} \tag{1.18}$$

L'entrée et les sorties n'ont pas changées.

L'espace d'état, d'ordre 3, vaut :

$$\begin{cases} \dot{x}(t) = \begin{pmatrix} -\frac{R}{L} & 0 & -\frac{K_e}{L} \\ 0 & 0 & 1 \\ \frac{K_c}{J_2} & 0 & \frac{\mu}{J_2} \end{pmatrix} x(t) + \begin{pmatrix} \frac{1}{L} \\ 0 \\ 0 \end{pmatrix} u(t) \\ y(t) = \begin{pmatrix} 0 & 0 & K_g \\ 0 & K_r K_s & 0 \end{pmatrix} x(t) \end{cases} \tag{1.19}$$

1.5 Espace d'état d'ordre 2, modèle EE2

Pour correspondre avec un modèle de comportement, nous allons à nouveau réduire la représentation d'état (1.17) pour obtenir un modèle d'ordre 2. Pour cela, nous allons annuler l'effet des dynamiques des courants i_1 et i_2 qui sont beaucoup plus grandes que les dynamiques de ω et θ, qui sont celles que nous sommes capable de mesurer et que nous souhaitons asservir. Ainsi nous avons un espace d'état où il sera plus facile de concevoir un retour d'état. Le système d'ordre 3 servira donc de modèle permettant une validation intermédiaire entre celui d'ordre 2 et celui d'ordre 4.

Le vecteur d'état vaut maintenant :

$$x(t) = \begin{bmatrix} \Omega_m \\ \Theta_m \end{bmatrix} \tag{1.20}$$

Le modèle espace d'état d'ordre 2 vaut :

$$\begin{cases} \dot{X}(t) = \begin{pmatrix} 0 & 1 \\ 0 & -\frac{(K_c K_e)}{J_2 R} - \frac{\mu}{J_2} \end{pmatrix} X(t) + \begin{pmatrix} \frac{K_c}{J_2 R} \\ 0 \end{pmatrix} V_m(t) \\ Y(t) = \begin{pmatrix} 0 & K_g \\ K_r K_s & 0 \end{pmatrix} X(t) \end{cases} \tag{1.21}$$

2 Analyse

Dans ce chapitre, nous allons dans une première partie, étudier la stabilité de nos différentes modélisations (espace d'état d'ordre 4, 3 et 2), puis leurs commandabilités et observabilités. Dans une seconde partie, nous étudierons les performances dynamiques des différents modèles à travers une analyse temporelles et fréquentielle. Dans l'ensemble du chapitre sera abordé l'impact des simplifications effectués sur les modèles espace d'état d'ordre 3 et 2. Comme nous souhaitons asservir le procédé en vitesse et non en position, nous étudierons la sortie de performance $V_g(t)$ des modèles et non $V_s(t)$.

2.1 Analyse des modèles

2.1.1 Stabilité

Nous avons décidé d'étudier la stabilité asymptotique afin de savoir si l'ensemble des états de nos modèles sont stables et non uniquement ceux qui sont observables comme en stabilité BIBO.

Les valeurs propres de notre système d'ordre 4 sont : (calculé à l'aide de matlab)

$$0; -132749,8861; -4,0655; -7748,0483 \tag{2.1}$$

Nous remarquons que les valeurs propres sont toutes à partie réelle négatives, le système d'ordre 4 est donc asymptotiquement stable.
Les valeurs propres de notre système d'ordre 3 sont : (calculé à l'aide de matlab)

$$0; -7748,0484; -3,9516 \tag{2.2}$$

Nous remarquons que les valeurs propres sont toutes à parties réelles négatives, le système d'ordre 3 est donc asymptotiquement stable.

C'était prévisible car le modèle d'ordre 3 est une simplification du modèle d'ordre 4, qui est stable. Nous remarquons aussi que la troisième valeur propre du système d'ordre 3, qui normalement doit être similaire à la troisième valeur propre du système d'ordre 4 a légèrement variée. Cette différence est une première conséquence de la perte d'une dynamique engendrée par la simplification.
Les valeurs propres de notre système d'ordre 2 sont : (calculé à l'aide de matlab)

$$0; -3,9506 \tag{2.3}$$

Nous remarquons que les valeurs propres sont toutes à parties réelles négatives, le système d'ordre 2 est donc asymptotiquement stable.
Comme précédemment, cette conclusion était prévisible, néanmoins on remarque une autre conséquence de la simplification sur la seconde valeur propre qui est légèrement différente de celle du modèle d'ordre 3.

2.1.2 Commandabilité

L'étude de la commandabilité d'un système nous permettra de savoir quels états sont commandables, c'est à dire qu'il sera possible de modifier la dynamique qu'ils représentent par un asservissement. Cette étude se fera uniquement sur l'espace d'état d'ordre 4 car les deux autres modèles découlent de celui-ci. Un système (sous forme d'espace d'état) est commandable, d'après le critère de Kalman, si la matrice de commandabilité $\mathcal{C}_m$ est de rang plein donc égale à la dimension de A.

Où, $\mathcal{C}_m = \begin{bmatrix} B & AB & \dots & A^{n-1}B \end{bmatrix}$ Nous avons étudié la commandabilité sur matlab et le résultat est que :

$$rang(\mathcal{C}_m) = n = 4$$

Donc le système est commandable. Cela nous permet de mettre en place un retour d'état.

2.1.3 Observabilité

L'étude de l'observabilité d'un système nous permettra de savoir quels états sont observables, c'est à dire s'il est possible de déterminer la valeur des états à partir de mesures de la sortie. Cette étude se fera uniquement sur l'espace d'état d'ordre 4 car les deux autres modèles découlent de celui-ci. Un système (sous forme d'espace d'état) est observable, d'après le critère de Kalman, si la matrice d'observabilité $\mathcal{O}$ est de rang plein donc égale à la dimension de A.

Où, $\mathcal{O} = \begin{bmatrix} C \\ CA \\ \vdots \\ CA^{n-1} \end{bmatrix}$ Nous avons étudié l'observabilité sur matlab et le résultat est que :

$$rang(\mathcal{O}) = dim(A) = 4$$

Donc le système est observable, néanmoins intuitivement ce résultat semble faux.

2.2 Analyse temporelle et fréquentielle

Nous avons étudié les performances de nos modèles grâce à deux types de réponses :

— Une réponse à un échelon unité (voir figure 2.1).

— Une réponse fréquentielle représentée par un diagramme de Bode (voir figure 2.2).

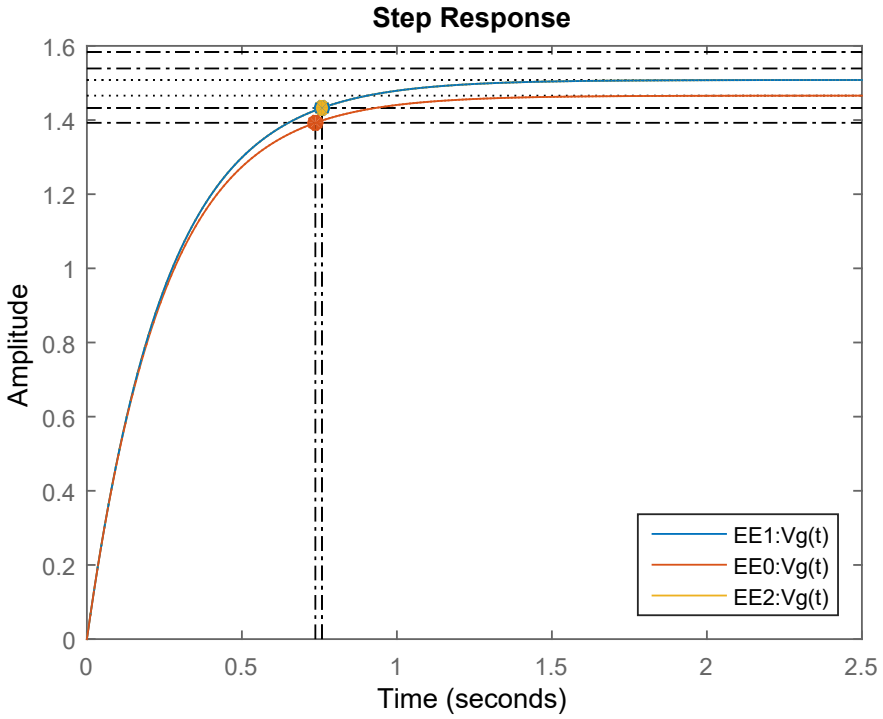


FIGURE 2.1 – Réponse à un échelon unité de $V_g(t)$ des modèles EE0, EE1 et EE2.

2.2.1 Performances Statiques

Nous étudions $V_g(t)$ en temps que sortie de performance (figure 2.1).
Nos modèles présentent des gains statiques qui varient d'un modèle à l'autre, à cause des simplifications :

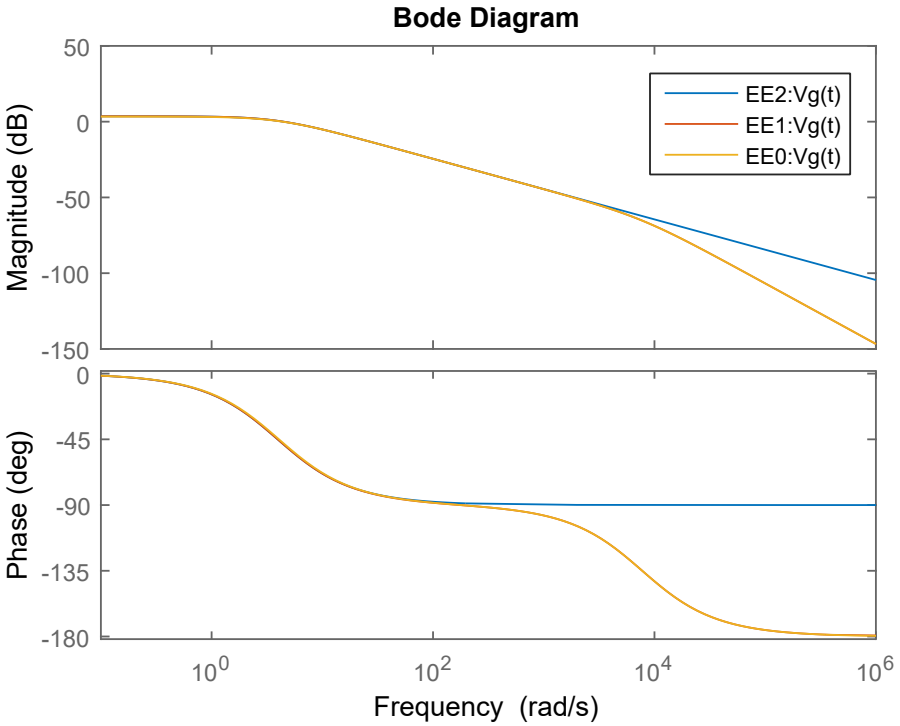


FIGURE 2.2 – Diagramme de Bode sur $V_g(t)$ des modèles EE0, EE1 et EE2.

Gain statique de EE0 : $1,4658$

Gain statique de EE1 : 1.5080

Gain statique de EE0 : 1.5080

On peut constater qu'entre le modèle EE0 et le modèle EE1, il y a une erreur de 0.0423 et qu'en le modèle EE1 et le modèle EE2 l'erreur vaut $2.0241 * 10^{-8}$. La première simplification engendre une erreur d'environ 2% et la suivante de le l'ordre de 10^{-8}%. L'asservissement risquera donc de présenter une erreur statique causée par ces simplifications.

2.2.2 Performances dynamiques

Nous étudions $V_g(t)$ en temps que sortie de performance (figure 2.1).
Nous avons choisis d'étudier le temps de montée, le temps de réponse et le dépassement car ce sont des paramètres précis et déterminants en terme de performances.
Temps de montées :
Sur EE0 : 0.5404s
Sur EE1 : 0.5560s
Sur EE2 : 0.5561s
Erreur de EE0 à EE1 : 0.0156 s (2.8826%)
Erreur de EE1 à EE2 : 1.3830e-04 s(0.0249%)
Nous pouvons remarquer que le temps de montée est aux alentours d'une demie seconde. Les différences entre les temps de montées des différents modèles risquent de créer une erreur sur l'asservissement, en effet elle témoigne d'une différence dans les dynamiques des modèles (ce qui est une conséquence logique de la simplification). Le test de la commande sur EE0 risque de démontrer une erreur dynamique.

Temps de réponses à 5% :
Sur EE0 : 0.7370 s
Sur EE1 : 0.7582 s
Sur EE2 : 0.7583 s
Erreur de EE0 à EE1 : 0.0212 s (2.8821%)
Erreur de EE1 à EE2 : 5.9211e-05 s(7.8089e-05%)

Comme pour la performance précédente , la valeur du temps de réponse varie et cela entrainera de potentielles erreurs d'asservissement.
Dépassements :
Le modèle en boucle ouverte ne présente pas de dépassement.

2.2.3 Analyse fréquentielle

On peut remarquer sur le diagramme de Bode, figure 2.2, qu'il y a une compensation pôle zéro dans le modèle EE0 car nous avons uniquement trois changements d'allures. Cela se confirme sur une vue des pôles et des zéros sur un plan complexe (voir figure 2.3) ou une fonction de transfert. Cette figure met en évidence la compensation d'un pôle électrique qui agis à haute fréquence. Sur le diagramme de Bode, nous pouvons aussi constater que les modèles semblent avoir des réponses fréquentielles assez similaires. On peut aussi voir l'absence de la dernière dynamique de EE2 due à la simplification.

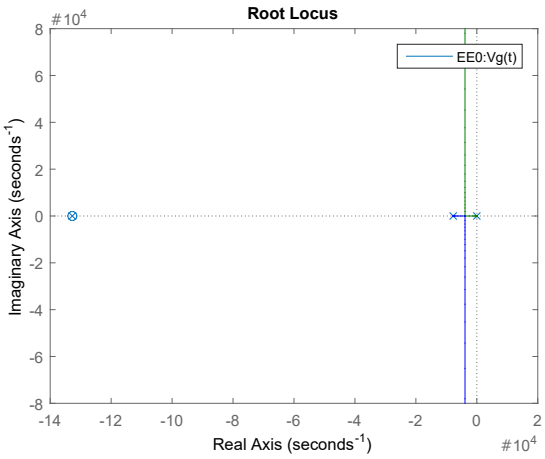


FIGURE 2.3 – Pôles et zéros dans le plan complexe du transfert vers $V_g(t)$ du modèles EE0.

3 Synthèse de commande

3.1 Commande du système d'ordre 2

3.1.1 Rédaction du cahier des charges et démarche de réponse

Après l'étude que nous venons de réaliser sur notre système, nous allons ici exprimer les attentes que doit réaliser la commande que nous allons implémenter. Nous souhaitons avoir :

— Erreur de position nulle.

— Pas d'oscillations

— Temps de réponse inférieur à 1 seconde.

La commande de notre système doit permettre d'asservir le système en vitesse, par rapport à une consigne. Pour le respecter, nous allons réaliser un placement de valeurs propres par retour d'état.

3.1.2 Calcul du retour d'état

Pour garantir les performances dynamiques souhaitées, nous allons appliquer un placement de pôles par retour d'état. Ce réajustement des valeurs propres de la matrice dynamique du système nous permettra de répondre aux attentes du cahier des charges si le choix de celles-ci est correct. De même, nous devons choisir des valeurs propres qui ne sont pas être trop éloignées de celles du procédé, pour ne pas être trop exigeant avec la commande et le système. Nous avons, avec ces spécifications, choisi les valeur propres suivantes :

$$\lambda = \begin{pmatrix} -4 & -5 \end{pmatrix} \tag{3.1}$$

La commande par retour d'état est une méthode d'asservissement qui permet d'effectuer un placement de pôles. Cette correction se réalise grâce à un gain K qui va corriger l'état x d'un système. Cette loi de commande s'écrit :

$$u(t) = Ny_{ref} - Kx(t) \tag{3.2}$$

avec N un gain de pré-compensation, y_{ref} la vitesse de référence, $x(t)$ la sortie mesuré du système et K le gain de retour. Si l'on applique cette loi à notre système en espace d'état, on obtient :

$$\begin{cases} \dot{X}(t) = (A - BK)X(t) + BNy_{ref} \\ Y = CX(t) \end{cases}$$

Ainsi la nouvelle dynamique du système est donnée par la matrice $A' = (A - BK)$ et doit admettre les valeurs propres que nous désirons appliquer à notre système. A et B étant des paramètres du système, nous allons utiliser le gain K pour répondre à ce problème. Avec la fonction *place* de MATLAB, nous sommes capable de concevoir ce vecteur K.

3.1.3 Observateur ordre plein sur modèle d'ordre 2

Pour pouvoir réaliser notre commande par retour d'état, nous devons tous d'abord reconstruire l'ensemble des états du système dont nous n'avons pas accès. Dans notre cas, nous disposons d'une mesure de la position du moteur θ avec V_s mais pas de la vitesse du moteur, ω, qui est nécessaire pour asservir le moteur.

Nous préférons reconstruire les deux états Ω et θ à partir de V_s et de l'entrée du modèle d'ordre 2 pour simplifier les calculs nécessaire à sa construction. Les équations de l'observateur sont présentées par :

$$\begin{cases} \dot{z}(t) = Fz(t) + Gy(t) + Hu(t) \\ \hat{x}(t) = Mz(t) + Ny(t) \\ \epsilon(t) = x - \hat{x} \end{cases}$$

où x représente l'état du système, $\hat{x}$ l'état du système reconstruit et ϵ l'erreur d'estimation à un temps t. Nous souhaitons contrôler la dynamique de ce paramètre pour pouvoir estimer correctement l'état de notre système. Pour cela, nous nous intéressons à :

$$\dot{\epsilon} = \dot{x} - \dot{\hat{x}} \tag{3.3}$$
$$\Leftrightarrow \quad \dot{\epsilon} = Ax + Bu - F\hat{x} - Gy - Hu \text{ en considérant } M = 1 \text{ et } N = 0 \tag{3.4}$$
$$\Leftrightarrow \quad \dot{\epsilon} = Ax - F\hat{x} - GCx + u(B - H) \tag{3.5}$$
$$\Leftrightarrow \quad \dot{\epsilon} = (A - GC - F)x + F\epsilon + u(B - H) \tag{3.6}$$

Il vient alors $F = A - GC$ et $B = H$ pour obtenir $\dot{\epsilon} = F\epsilon$. Ainsi l'erreur d'estimation est autonome et ne dépend pas des entrées et sorties du système, et il vient $\epsilon(t) = e^{Ft}\epsilon(0)$. Les valeurs propres de F vont ainsi déterminer la dynamique de l'erreur d'estimation, nous choisissons de les faire dépendre des valeurs propres désirées dans la partie 3.1.2 en les multipliant par 3, pour une convergence encore plus rapide.

$$v_{p_{obs}} = 3 \times \lambda \tag{3.7}$$

3.1.4 Construction de l'asservissement

Maintenant que l'observateur et le gain du retour d'état sont construits, nous allons les réunir pour former un unique modèle de commande. Pour ce modèle du système bouclé, nous prenons comme vecteur d'état $X = \begin{pmatrix} X & \epsilon \end{pmatrix}^T$ et nous obtenons par construction :

$$\begin{cases} \dot{X} = \begin{pmatrix} (A - BK) & -BK \\ 0 & F \end{pmatrix} X + \begin{pmatrix} BN \\ 0 \end{pmatrix} y_{ref} \\ y(t) = \begin{pmatrix} C & 0 \end{pmatrix} X \end{cases}$$

Le gain de pré-compensation N est à partir gain statique du transfert entre Vg et y_{ref} noté $G_{Vg/y_{ref}}$ pour permettre une erreur de position nulle. Il est :

$$N = \frac{1}{G_{Vg/y_{ref}}} \tag{3.8}$$

Cependant, pour permettre un asservissement du système en vitesse ω, nous devons modifier légèrement le résultat obtenu. La description telle qu'elle est donnée du système bouclé va asservir le système en position, à cause du changement de dynamiques de la position θ avec le retour d'état. Pour prévenir à cette mauvaise commande, nous avons annulé le retour de la mesure de position dans le système bouclé.

A partir de ce modèle, nous allons être capable d'éprouver la commande obtenu de manière théorique et avec une simulation SIMULINK.

3.2 Validation de la commande

3.2.1 Validation théorique

Pour commencer, nous allons identifier les valeurs propres de notre système afin de connaitre les effets de l'observateur sur le retour d'état, même si à priori celui ci est autonome. Nous obtenons les λ_i valeurs propres suivantes : -5, -4, -15 et -12. Nous remarquons que les λ_i n'ont pas été modifiées : nous avons les valeurs propres désirées par la commande et nous avons celles désirées pour l'erreur d'estimation de l'observateur.

3.2.2 Simulation *SIMULINK*

Pour compléter la validation de la commande, nous avons crée un prototype avec *SIMULINK* pour pouvoir simuler une réponse temporelle de la commande de notre système. La description du fichier se trouve en annexe 7. Nous observons sur les figures 3.1 la réponse temporelle à une référence $y_{ref} = 1$: La réponse du système Vs simulé correspond aux changement de dynamique que nous avons espéré. De plus, après une observation de la commande $u(t)$ appliquée sur le système, nous voyons que la tension maximale envoyée est borné dans l'ordre de grandeur des valeurs admises par le système physique.

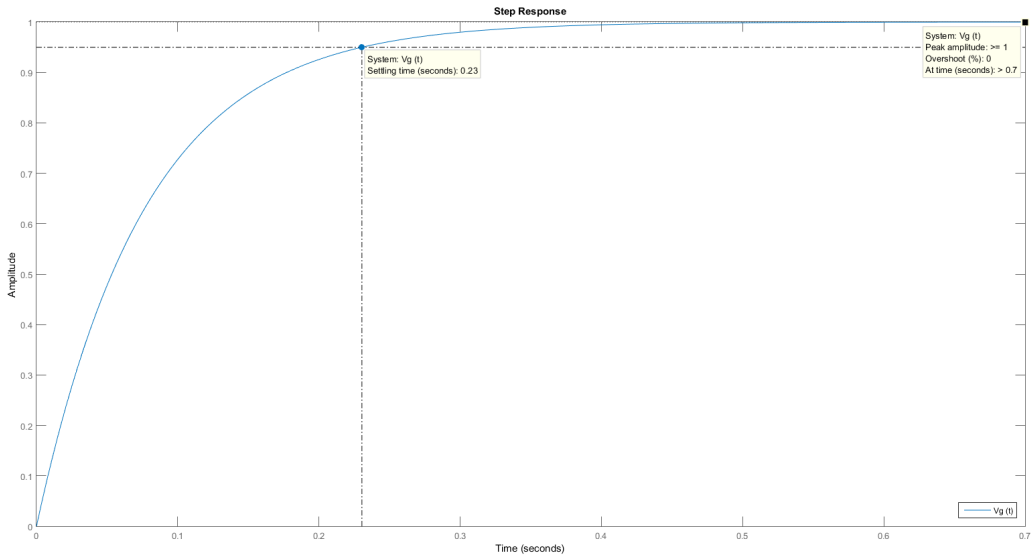


FIGURE 3.1 – Réponse du système asservi

3.2.3 Analyse boucle fermé du modèle d'ordre 3

Notre commande respecte le cahier des charges, nous validons ainsi la commande pour notre modèle linéaire d'ordre 2. Nous allons maintenant analyser cette même commande sur des modèles supérieur et plus complexes pour compléter sa validation. En effet, notre modèle a été très simplifié pour permettre la création d'une commande facilement. Si nous arrivons à prouver que cette commande marche sur des systèmes plus complexes, alors nous aurons prouvé que les dynamiques que nous avons simplifiés ne vont pas perturber notre commande.

3.3 Validation sur les modèles d'ordre supérieur

3.3.1 Changement de base du modèle d'ordre 3

Dans un premier temps, nous regarderons les effets de l'implantation de l'observateur d'ordre 2 sur les modèles d'ordre supérieur et plus précisément l'erreur d'estimation et le placement des valeurs propres, puis nous simulerons la commande obtenu pour observé les modifications des réponses.

Pour intégrer l'observateur dans un espace d'état d'ordre supérieur, nous avons du réorganiser les états du modèle, ici le modèle d'ordre 3, de façon à ce que les états reconstruits avec l'observateur soient en haut, x_1, et les non reconstruits en bas x_2.

- États reconstruits : Ω_m et Θ_m.
- États non reconstruits : i_1

On obtient le vecteur d'état suivant :

$$\overline{X} = \begin{bmatrix} \theta \\ \omega \\ i_1 \end{bmatrix} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \tag{3.9}$$

donc $x_1 = \begin{bmatrix} \theta \\ \omega \end{bmatrix}$ et $x_2 = i_1$. Pour passer de X à $\overline{X}$, il faut utiliser une matrice de passage P_X définit par :

$$P_X \quad / \quad \overline{X} = P_X \cdot X \tag{3.10}$$

$$P_X \quad = \quad \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \tag{3.11}$$

$$\tag{3.12}$$

Maintenant, nous devons calculer $\overline{A}$, $\overline{B}$, $\overline{C}$, $\overline{D}$ pour le nouvel espace d'état lié à $\overline{X}$.

$$\begin{cases} \overline{A} &= P^{-1}AP \\ \overline{B} &= P^{-1}B \\ \overline{C} &= CP \end{cases} \tag{3.13}$$

3.3.2 Analyse du changement de base et de l'erreur de reconstruction

Maintenant que nous avons replacé le modèle d'état, nous pouvons commencer l'analyse de celui ci. Avec les résultats obtenus pendant les séances de TD et de cours, nous avons obtenu une relation de la dynamique de l'erreur

$$\dot{\epsilon} = F\epsilon - A_2 x_2 \tag{3.14}$$

avec A_2 la partie de la matrice A qui lie l'état $\dot{x}_1$ à x_2 dans la nouvelle base. Cet élément modifie les résultats que nous avons obtenu pour le modèle d'ordre 2 en indiquant que l'évolution de l'erreur de reconstruction des états n'est plus en fonction uniquement d'elle même, mais aussi des états qui n'ont pas été reconstruits. Cette nouvelle donné va perturbé la convergence de la reconstruction de l'erreur, nous n'arriverons pas à reconstruire l'ensemble des états.

Nous allons étudier les nouvelles valeurs propres du système bouclé pour vérifier si celles si correspondent toujours au cahier des charges, en modélisant un espace d'état qui admet comme vecteur d'état $x = \begin{pmatrix} \overline{X} & \epsilon \end{pmatrix}^T$. Nous obtenons : -7748, -20.61, -3.832 ± 6.931 et -2.784, la stabilité asymptotique en boucle fermé est toujours respectée.

Pour continuer l'analyse de cette commande, nous obtenons à partir de tracé *MATLAB* les réponses temporelles du système en boucle fermé avec les caractéristiques suivantes :

— Le gain statique n'est plus respecté. Nous commençons à voir apparaitre une erreur de position en régime statique. Cette erreur est due à l'erreur de reconstruction de

— le temps de réponse reste dans la clause du cahier des charges et pas d'oscillations

Pour finir l'analyse de la boucle fermé, nous allons étudier le transfert de $V_g(t)$ avec la consigne et le comparer avec celui du modèle d'ordre 2. Nous obtenons le diagramme de Bode en figure 3.2.

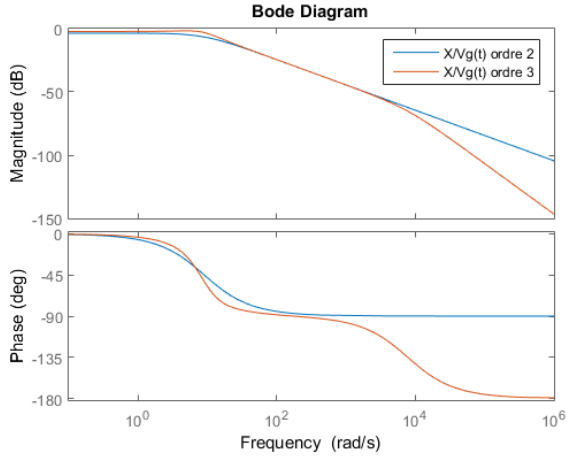


FIGURE 3.2 – Transfert de la boucle fermé, modèle linéaire d'ordre 3

Le résultat que nous obtenons confirmes la stabilité asymptotique établie précédemment, cependant les différences notables avec le transfert du modèle d'ordre 2 en basse fréquence explique pourquoi nous n'avons pas pu respecter exactement le cahier des charges.

3.3.3 Changement de base du modèle d'ordre 4 et intégration de la commande

De la même manière que pour le modèle d'ordre 3, nous effectuons un changement de base avec la matrice de passage :

$$P = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \tag{3.15}$$

Nous obtenons ainsi comme dans la section précédente, un modèle d'ordre 4 qui a comme vecteur d'état :

$$\overline{X} = \begin{pmatrix} \theta & \omega & i_1 & i_2 \end{pmatrix}^T \tag{3.16}$$

Les performances du système bouclé ont les mêmes problèmes que pour le modèle d'ordre 3. Les reconstructions des états ne convergent pas vers 0, comme nous l'avions vu précédemment, ce qui perturbe l'asservissement en vitesse.

De plus, les approximations de modélisation ont montré dans l'analyse des modèles une erreur statique que nous ne pouvons pas corriger avec une commande construite sur le modèle d'ordre 2. Nous avons toutefois voulu analyser le transfert entre Vg et la consigne et obtenons : Nous avons de plus amples écart en basse fréquence

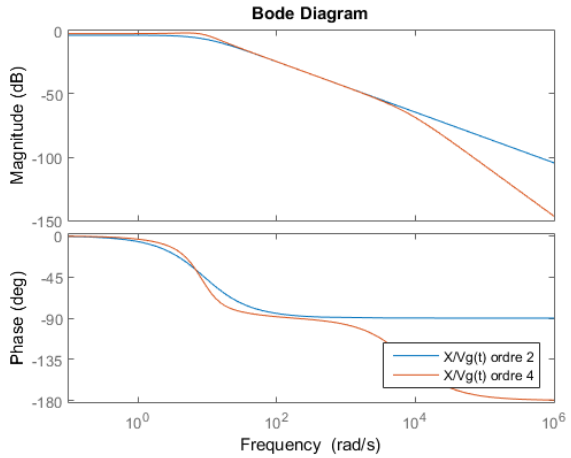


FIGURE 3.3 – Transfert de la boucle fermé, modèle linéaire d'ordre 4

avec un dépassement encore plus grand que pour le modèle d'ordre 3. Ce dépassement joue beaucoup sur les oscillations du système en boucle fermé. Le cahier des charges n'est encore une fois pas respecté pour ce modèle mais admet un temps de réponse proche de celui du modèle d'ordre 2.

En sachant ceci, nous savons que les dynamiques qui ont été simplifiés ne sont pas l'origine des écarts que nous venons d'exposer, il s'agit de la dynamique de l'observateur qui n'est pas optimale.

3.3.4 Étude de la réponse entre l'erreur de reconstruction par rapport à une commande

Avec les changements de base calculé précédemment, nous avons souhaité analysé les transferts entre l'erreur de reconstruction ϵ et la commande u. Pour cela, nous avons utilisé une réponse temporelle affichée en figure 3.4. Dans cette figure, nous voyons que l'erreur pour le modèle $EE2$ d'ordre 2, l'erreur converge très rapidement vers 0. Pour les autres modèles, l'erreur converge aussi, mais pas vers 0. Ce décalage du régime statique de ϵ ne nous permettra pas de reconstruire parfaitement l'état de la vitesse du moteur, mais uniquement de nous en approcher.

3.4 Simulation $SIMULINK$ des modèles linéaires

Maintenant que l'observateur est prêt à être implémenter sur les modèles d'ordre 3 et 4, nous allons pouvoir commencer à exécuter des simulations des commandes de ces deux modèles. Deux choix s'offrent à nous, nous pouvons utiliser, dans un premier cas, le changement de base et donc un système en espace d'état de notre

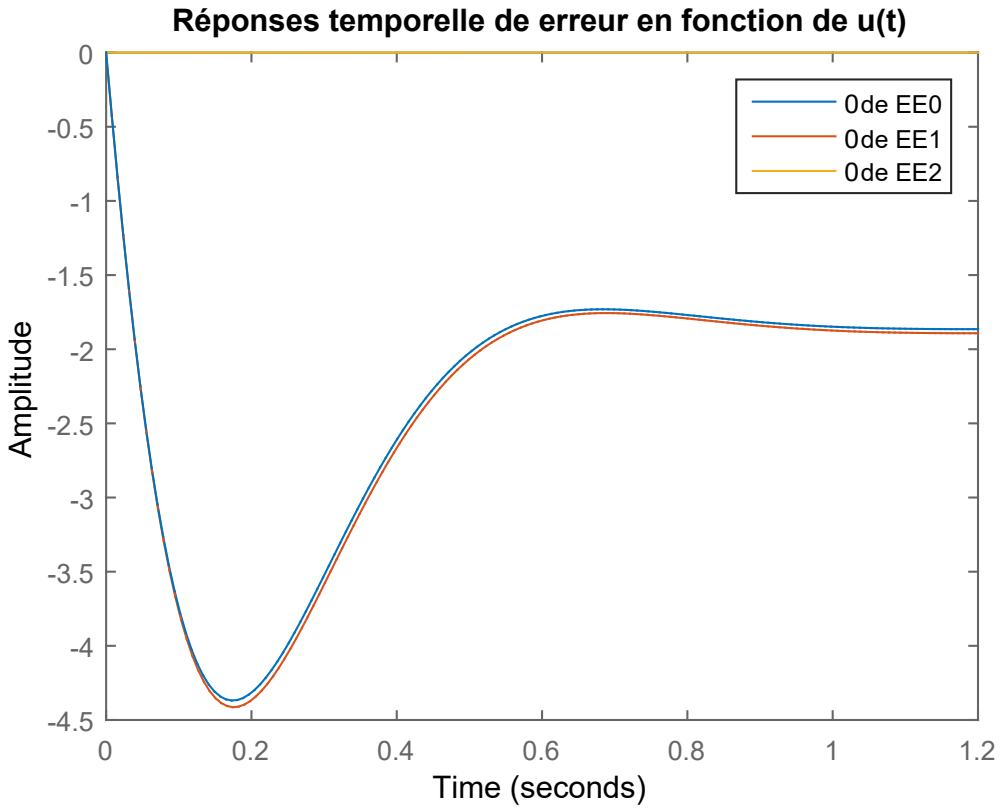


FIGURE 3.4 – Réponses de l'erreur de reconstruction de l'état ω(vitesse) des modèle d'ordre 2 à 4

système bouclé. Dans cette situation, nous aurions accès à la vitesse asservis en fonction d'une référence. Le deuxième choix, celui qui nous semble le plus judicieux, consiste à brancher chaque élément sur un schéma $SIMULINK$. Nous avons ainsi accès à tous les échanges de signaux entre tous les blocs.

Les schémas blocs que nous avons simulés sont disponibles en annexe 7. La réponse temporelle de $V_g(t)$ se trouve, pour les modèles linéaires d'ordre 2 à 3, dans la figure 3.5, ainsi que la commande qui est associé à cette réponse, en figure 3.5.

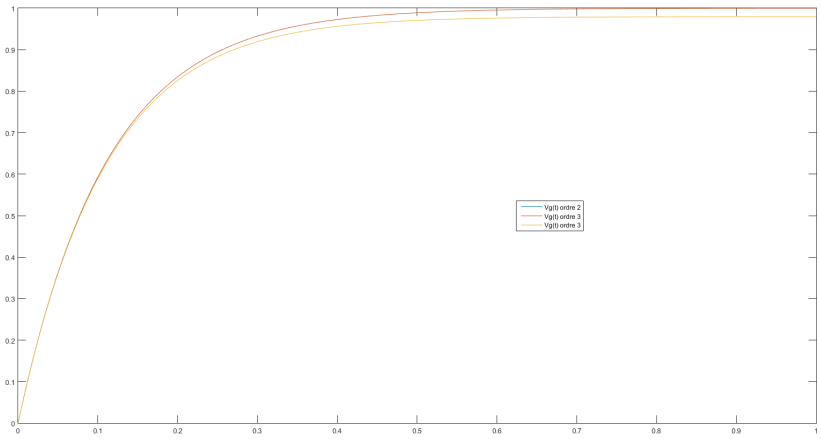


FIGURE 3.5 – Réponses temporelles des boucles fermées sous $SIMULINK$

Sur ces figures, pour le modèle linéaire d'ordre 4, nous avons obtenu le bon temps de réponse et pas d'oscillations. L'erreur du régime permanent est de 2%. Ces résultats nous permettent de regarder une simulation des signaux de $Vg(t)$. D'après ces résultats, la commande que nous souhaitons appliquer respecte le cahier des charges en termes d'oscillations de la réponse et de temps de réponse mais pas pour le gain statique. Cependant,

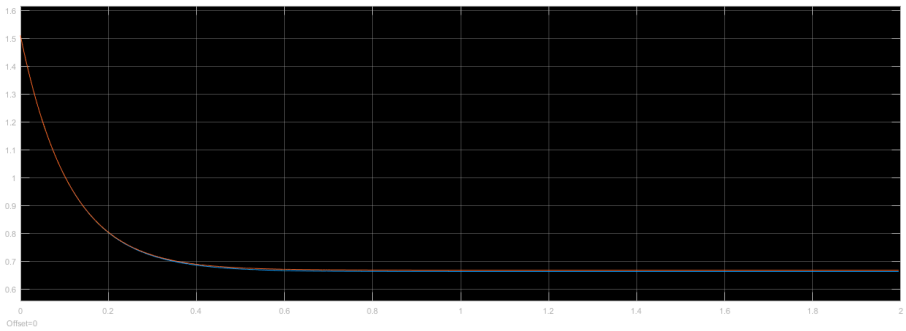


FIGURE 3.6 – Commandes $u(t)$ associées aux résultats de la figure 3.5

nous ne pouvons en tirer aucune conclusion , il reste encore d'autres modèles plus complexes sur lequel notre commande doit être tester.

3.5 Conclusion

Nous avons maintenant terminé les exemples de simulations sur les modèles linéaires du moteur que nous avons calculé. A partir de cet instant, nous avons deux possibilités, nous pouvons tester directement le retour d'état sur le moteur, sans vraiment connaitre les conséquences du passage de notre modèle linéaire au procédé réel du moteur, soit nous analysons d'abord la loi de commande créer ici, sur un modèle non linéaire. Nous aborderons cette dernère proposition dans le chapitre prochain.

4 Validation de la commande en temps continue sur le modèle non linéaire

Maintenant que les étapes de démonstration et de simulation de la commande sur des modèles linéaires sont terminés, nous allons pouvoir créer notre premier prototype, que nous nommerons prototype 0. Il correspondra à notre solution 0. Nous allons d'abord effectuer ce que nous appelons un *Model in the loop* où nous allons chercher à améliorer ce premier prototype en utilisant uniquement des simulations. Par la suite, nous améliorerons le prototype en utilisation une technique d'émulation dite *Software in the loop*.

4.1 Protocoles MIL/SIL

Dans cette partie, nous allons expliquer les démarches que nous utiliserons pour valider nos prototypes.

4.1.1 Model in the loop (MIL)

Pour faire une validation MIL, nous allons utiliser le modèle non linéaire pour simuler la correction. Toute cette opération sera effectué sous *MATLAB* à l'aide d'un bloc *SIMULINK* qui simule le modèle non linéaire. Pour valider le prototype de commande du correcteur, celui-ci devra respecter la ou les condition que nous lui imposerons. S'il ce n'est pas le cas, une amélioration de ce prototype sera nécessaire et elle nous permettra de créer un prototype N (pour N itérations de cette boucle).

4.1.2 Software in the loop (SIL)

Après la validation en MIL, nous passerons à une validation *Software in the loop*. Le prototype obtenu au bout de la boucle MIL sera ici testé sur sa partie logicielle. Nous testerons ici si le temps concret de commande est vérifié, en branchant directement le calcul de la commande sur un autre ordinateur ou bien en utilisant un moteur réel. Pour cela, nous n'hésiterons pas à effectuer un prototypage rapide, une fonction de *MATLAB*, qui permet de recevoir et d'émettre des signaux analogiques depuis un ordinateur via un une carte E/S.

De même que pour le MIL, si le prototype N ne valide pas le cahier des charges, nous devons recommencer le test en créant un nouveau prototype $N+1$ jusqu'à que la commande soit satisfaisante.

A présent, nous pouvons commencer à tester les prototypes de commandes du moteur sur des modèles plus complexe : nous disposons pour cela d'un modèle non linéaire ainsi que d'un banc moteur. Nous suivront les consignes que nous venons de présenter en 4.1, en commençant par présenter le moyen utilisé pour adapter la simulation. Comme il est évoqué dans le titre de ce chapitre, nous sommes ici en temps continu. Les hypothèses pour la discrétisation ne sont pas traités dans cette partie, nous avons choisi de construire le prototype de commande à temps discret quand nous aurons d'abord validé complètement la commande en temps continu, et que nous auront présenté toutes les contraintes que la discrétisation devra surmonter.

4.2 Simulation sur Modèle Non linéaire

Nous disposons de la démarche que nous allons utiliser pour valider notre simulation. Pour cette partie, nous utiliserons un bloc *SIMULINK* qui nous permettra de simuler un modèle très proche du banc moteur. Ce modèle n'a pas reçu les simplifications que posent les hypothèses faites en 1.2.3 et en 1.2.4. Un bloc *SIMULINK* est disponible à cet effet, sous forme de *S-Function*. Il utilise la modélisation et les paramètres utilisés en 1.2.2 pour créer un système Entrées/Sorties beaucoup plus proche que le modèle linéaire d'ordre 4.

4.2.1 Adaptation modèle

Pour permettre l'utilisation de ce modèle, nous devons d'abord lancer une compilation de la *S-function*. Cette opération est nécessaire car la *S-function* est un code en langage C qui s'adapte à la simulation *SIMULINK*, et êrmet ainsi d'obtenir des performances en temps de calcul plus élevé qu'en bloc de simulation classique. Nous avons choisi d'en-capsuler tout le bloc de commande (Gain du retour d'état et reconstruction d'état) dans un bloc *sub-system* que nous détaillons en annexe 7. Ce sous bloc prend en entrée 2 signaux : la consigne de référence y_ref et la sortie mesuré du système $y_m = V_s$ (Position du moteur, cf chapitre 1), et calcule la commande qu'il émet sur le connecteur nommé u.

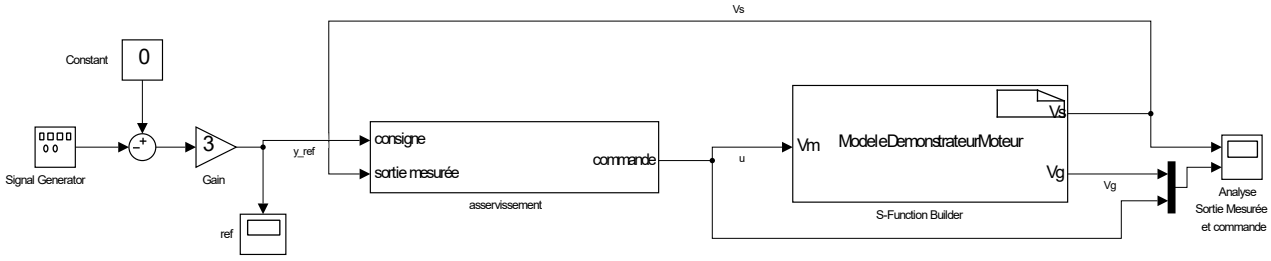


FIGURE 4.1 – Schéma *SIMULINK* complet de l'asservissement du modèle du moteur Non linéaire

Nous avons placé des points de mesure au sur le signal de référence ainsi qu'en sortie du système. Nous obtenons ainsi une réponse temporelle que nous allons étudier dans la sous section suivante.

4.2.2 Simulation et étude de performances

4.2.2.1 Nouveau paramètres non linéaires

Utilisons maintenant la simulation précédente pour étudier les performances de notre commande. Nous avons dans un premier temps observé la reconstruction d'état du système et nous avons pu noter un problème. Un premier problème apparait, nous avons un nouveau paramètre que nous n'avons pas modélisé dans les parties 1 et 2.

Cette nouvelle dynamique correspond à la mesure de la position du moteur : elle se retrouve être borné entre $[-5V; +5V]$ qui correspond aux limites des tensions générés par le banc moteur. Si elle dépasse l'un des seuils, elle est immédiatement retranscrite sur le seuil opposé. Nous pouvons représenter ce phénomène avec le système d'équation suivant :

$$V_s(t) = \begin{cases} K_r K_s \theta_s \text{ si } \theta_s \in \left[\dfrac{-5}{K_r K_s}; \dfrac{+5}{K_r K_s}\right] \\ -5 \text{ si } \theta_s < \dfrac{-5}{K_r K_s} \\ +5 \text{ si } \theta_s < \dfrac{+5}{K_r K_s} \end{cases}$$

La non linéarité de cette dynamique a été enlevé lors de la modélisation du système en modèle linéaire. Cela pose un problème pour la reconstruction de la vitesse par rapport à la position : nous avions remarqué en 3.3.3 et en 3.14 que lorsque l'observateur(pour l'ordre 2) était implémenté sur des modèle d'ordre supérieur, la dynamique de l'erreur de reconstruction n'était pas uniquement dépendante de lui même, les états non présents dans le modèle d'ordre 2, i.e. les 2 courants i_1 et i_2, ajoutent un décalage dans la reconstruction des états ω et θ.

4.2.2.2 Prototype 0

Avec tous ces éléments, nous sommes capables d'expliquer pourquoi les résultats obtenus pour les valeurs propres décidées dans les parties précédente donnent une reconstruction trop erroné de l'état du système. La reconstruction de la vitesse par rapport à la position qui est renvoyé vers l'opposé à chaque dépassement de $[-5; 5]$ avec les valeurs propres de l'observateur placé en $3 \times v_{p_{des}}$ (cf 3.1.3) donnent une restitution décalée, qui converge très mal et qui donc ne peut pas être utilisé pour calculer une commande. La non linéarité apparait, pour une consigne de vitesse de $3V$, à peu près 4 fois par seconde. L'observation de la réponse de ce prototype

est disponible sur la figure 4.2. Dans cette réponse temorelle, le gain K est donnée par le vecteur suivant :

$$K = \begin{pmatrix} 0 & 0.089 \end{pmatrix} \tag{4.1}$$

Pour rappel, nous annulons le retour de l'état de position pour asservir le moteur en vitesse uniquement. (cf 3.1.4). Nous observons sur cette figure que le changement de seuil de la tension V_s fausse totalement la

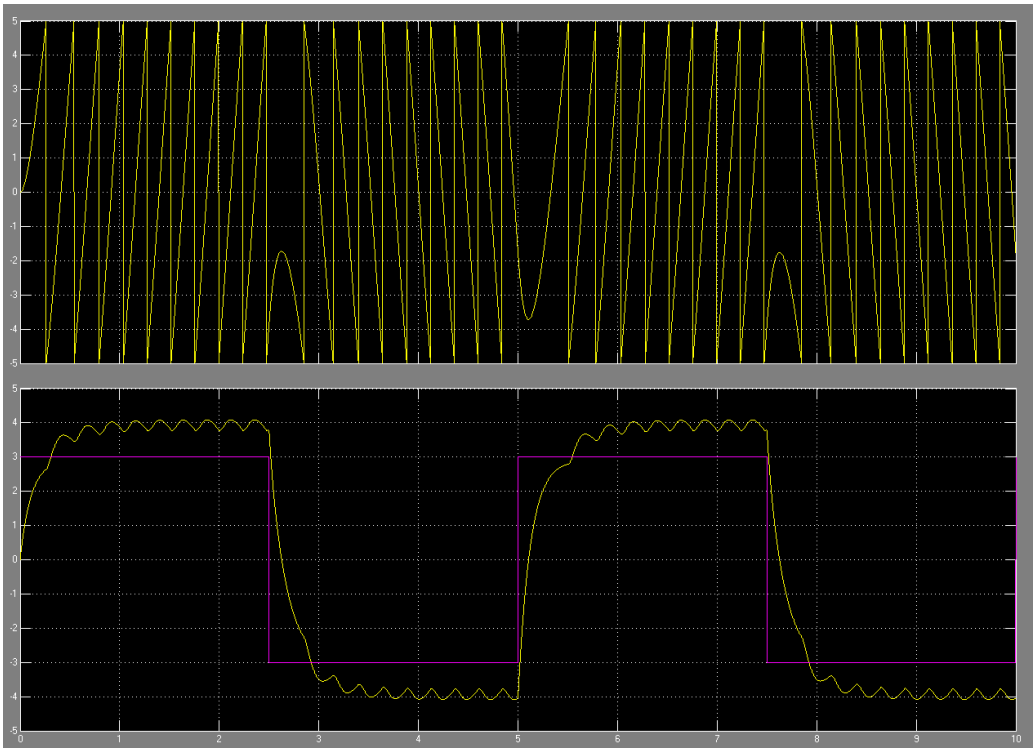


FIGURE 4.2 – Simulation réponse temporelle du modèle linéaire commandé par le prototype 0
Simulation des signaux V_s (Bloc du haut), de y_{ref}(Bloc du bas/courbe violet) et de V_m(Bloc du bas/courbe jaune)

reconstruction de la vitesse. Pour minimiser l'erreur sur ce modèle, nous devons faire coller les valeurs propres de l'observateur avec celles du système en boucle fermé. Cela devrait réduire les effets non linéaires de la sortie V_s. Nous choisissons de commencer l'étude d'un nouveau prototype.

4.2.2.3 Prototype 1

Nous avons changé les valeurs propres de l'observateur pour les faire correspondre avec les valeurs propres désirées.

Une nouvelle simulation faite sur ce nouveau prototype nous donne les résultats montrés dans la figure 4.3. Nous pouvons voir une erreur sur le régime permanent qui subsiste toujours. Ce léger décalage devrait pouvoir être corrigé sur un modèle linéaire avec un ajout d'un gain pour compenser ce manque. Or, nous sommes sur un modèle non linéaire et cette méthode ne peut pas fonctionner.

Un zoom sur l'erreur du régime permanent, disponoble en annexe 7 - figures 7.4 et 7.5, nous indque que pour une consigne $y_{ref} = 3$, nous obtenons $V_m(t_f) \approx 2.85$. Nous avons donc une erreur en régime statique

$$\epsilon_y = y_{ref} - V_m(t_f) = 0.15, \text{ qui correspond pour la consgne demandée à : } \epsilon = \frac{\epsilon_y}{y_{ref}} = 5\%$$

Avant de terminer l'analyse de ce prototype, nous nous intéressons au signal de commande envoyé sur l'émulation du moteur. Celle ci se trouve dans la figure 4.4.

La commande a une amplitude supérieur à celle de la référence, nous sommes donc dans un asservissement tel que la référence ne peux pas demander la vitesse maximale du moteur. En effet, si la référence demande une vitesse du moteur qui amène un signal de commande dépassant la tension maximale, la vitesse ne sera plus asservi par la commande, et nous serons dans un cas non déterminé où nous n'avons aucune possibilité de savoir si le cahier des charges sera respecté. Nous décidons de passer à l'implémentation d'un nouveau prototype. Ce prochain prototype devra corriger l'erreur de position ou du moins la minimiser et il devra vérifier que la commande ne dépasse pas le seuil maximal de la référence, pour un temps de réponse toujours dans les close du cahier des charges.

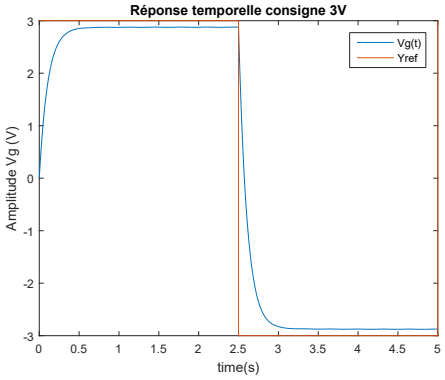


FIGURE 4.3 – Simulation réponse temporelle du modèle linéaire commandé par le prototype 1

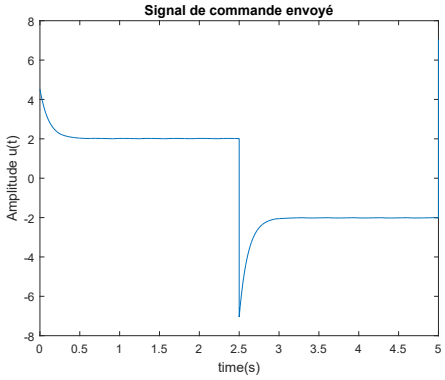


FIGURE 4.4 – Simulation signal de commande modèle non linéaire avec le prototype 1

Nous proposons de modéliser un nouveau prototype de commande en réutilisant le retour d'état et en ajoutant une action intégrale.

4.3 Retour d'état avec action intégrale

Ce nouveau prototype de commande va calculer un nouveau retour d'état, en utilisant cette fois ci la différence entre la référence et la sortie mesurée du système. Pour sa construction, nous nous référons au cours du Master ISTR EEA sur la construction d'un retour d'état.

Dans cet asservissement, un nouvel état du système noté x_i apparait dans la chaine directe. Il est décrit par la dynamique suivante :

$$\dot{x}_i = y_{mes} - y_{ref} \tag{4.2}$$

Ainsi, cet état dépend directement de la mesure et de la référence. Les équations du reste de l'espace d'état sont inchangés, et ainsi nous pouvons obtenir une nouvelle représentation d'état donnée avec comme vecteur d'état $X(t) = \begin{pmatrix} x & x_i \end{pmatrix} = \begin{pmatrix} \theta & \omega & x_i \end{pmatrix}$

$$\dot{X} = \begin{pmatrix} A & 0 \\ C & 0 \end{pmatrix} X(t) + \begin{pmatrix} B \\ 0 \end{pmatrix} u(t) + \begin{pmatrix} 0 \\ -1 \end{pmatrix} y_{ref} \tag{4.3}$$

Nous posons ensuite la nouvelle loi de commande :

$$u(t) = -K_i x_i(t) - K_p x(t) \tag{4.4}$$

Le gain K_i est le gain intégral associé à x_i et le gain K_p correspond au gain du retour d'état précédemment établi. Ces gains seront calculés comme un placement de pôle, avec la fonction *place*, une fois que les valeurs propres sont choisies. Pour celles ci, nous voulons appliquer la même dynamique que celle étudié en 3.1.2 en plus de la nouvelle (pour x_i). Les valeurs obtenu pour les gains sont les suivantes :

$$K_p = \begin{pmatrix} 0 & 0.5825 \end{pmatrix} \tag{4.5}$$

$$K_i = 28.5507 \tag{4.6}$$

Nous annulons encore le retour de la vitesse dans la chaine directe. Nous allons maintenant effectuer des simulations de ce nouveau retour sur le modèle linéaire. Une fois les simulations validés, nous pourrons passer à la validation de ce nouveau prototype sur un modèle non linéaire.

4.3.1 Simulation de l'action intégrale sur modèle linéaire et prototype sur modèle non linéaire

Les simulations sur modèle linéaire que nous avons effectué de ce retour d'état ont été faites uniquement sur le modèle d'ordre 4 qui est, de par son ordre, le plus complexe des modèle linéaire à notre disposition. Nous avons effectué les simulations avec l'environnement *SIMULINK*, le schéma du retour est disponible en annexe 7, figure 7.1. Dans ces schémas, nous n'avons pas mesuré la vitesse du moteur, nous l'avons reconstruit. Le signal de y_{mes} ne peut pas être la vitesse car inaccessible au mesure. Nous sommes donc obligés de reconstruire ce signal, via l'observateur en utilisant $\hat{x}$.

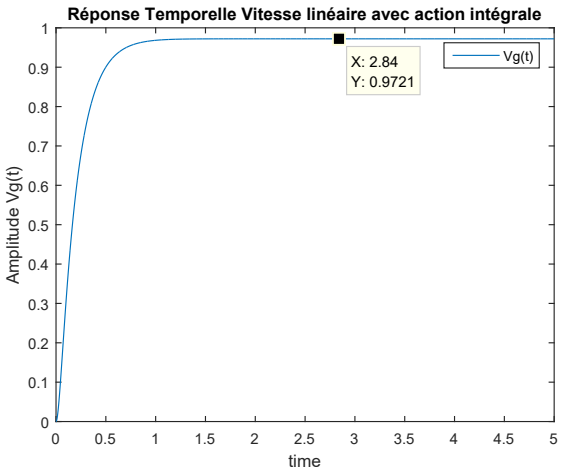


FIGURE 4.5 – Réponse temporelle du système linéaire asservis avec un retour d'état avec action intégrale

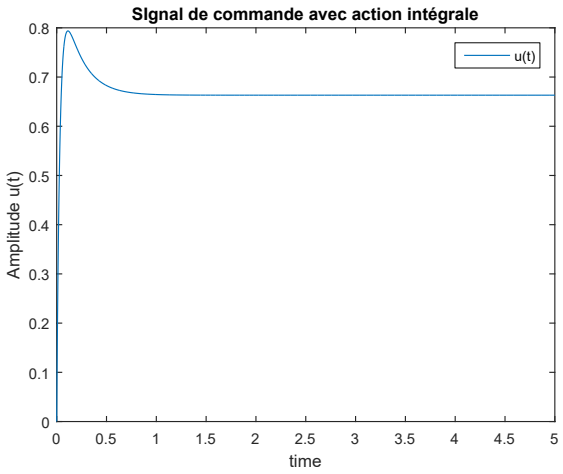


FIGURE 4.6 – Commande du système linéaire asservis avec un retour d'état avec action intégrale

Nous obtenons les courbes présentés en figures 4.5 et 4.6. Dans ces réponses simulés, nous observons toujours une erreur de position en régime statique.

$$\epsilon_y = y_{mes_{statique}} - y_{ref} = 0.03 \text{ donc : } \epsilon = 3\%$$

Comme nous le disions précédemment, le signal utilisé pour faire converger de l'erreur du régime statique est un signal reconstitué grâce à l'observateur. Cependant, il prend aussi les inconvénients de celui ci. Nous avions remarqué que l'observateur disposait d'une dynamique de reconstruction erroné(3.3.4), nous la retrouvons ici dans cette simulation. La commande appliquée est tout à fait convenable, elle ne dépasse pas la référence ce qui signifie qu'il est possible d'augmenter cette référence. Le temps de réponse n'a pas pu être mesuré correctement, cependant les observations de la figure 4.5 nous montrent que le régime statique est atteint quand la simulation dépasse les 1 seconde.

4.3.1.1 Prototype 2

Nous avons choisi de mettre cette nouvelle commande dans un nouveau prototype. Il est donc composé d'un retour d'état avec une action intégrale. Pour le valider, nous allons maintenant le simuler sur le même modèle non linéaire qui a été introduit pour les deux premiers prototypes. Nous pouvons observer les résultats sur les deux figures 4.7 pour le signal de performances $Vg(t)$ et 4.8 pour la commande $u(t)$ qui est envoyé dans le procédé non linéaire. Sur ces courbes, nous pouvons noter que l'erreur du régime statique est, cette fois ci,

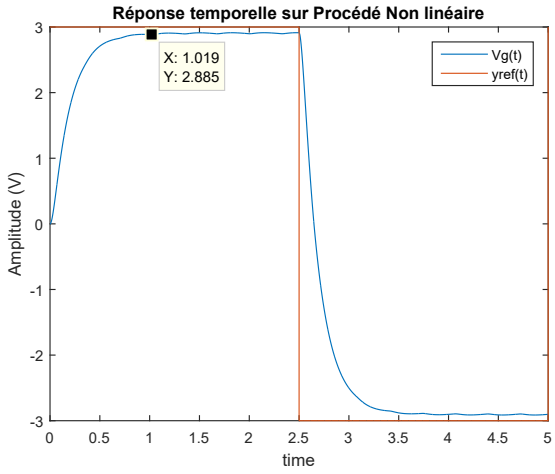


FIGURE 4.7 – Réponse temporelle du système Non linéaire asservis avec un retour d'état avec action intégrale

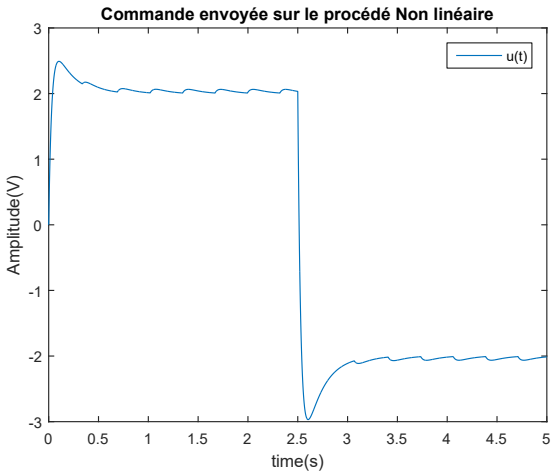


FIGURE 4.8 – Commande du système Non linéaire asservis avec un retour d'état avec action intégrale

$\epsilon_y = 3 - 2.885 = 0.115$, soit, $\epsilon = 3.8\%$. Cette valeur n'est pas inférieure à tous les prototypes testés jusqu'à

présent, cependant le signal de commande $u(t)$ ne dépasse pas y_{ref}, donc les limites d'amplitude de la commande ne seront pas dépassé. Le temps de monté est lui aussi respecté et nous n'avons pas d'oscillations. La validation de ce prototype n'est pas tout à fait donné, mais nous savons qu'avec l'erreur de reconstruction de ω, obtenir une erreur en régime statique non nul demande la création d'un modèle beaucoup plus perfectionné.

4.3.2 Conclusion et Validation

Les 3 prototypes qui ont sont ressortis du *Model in the loop* n'ont pas tous la même valeurs. Nous avons, pour résumé, un prototype 0 possède un observateur avec de mauvaises valeurs propres, et fait osciller la reconstruction des états. Nous disposons aussi d'un autre prototype qui lui contient une bonne dynamique en régime transitoire mais n'a pas un bon régime permanent. Enfin, le prototype 2, dispose aussi d'un très bon régime transitoire mais il n'attend pas l'idéal d'une erreur nulle en régime permanent.

4.4 Simulation sur moteur Réel

Nous souhaitons maintenant améliorer nos prototypes avec le banc moteur directement. En utilisant la fonction de prototypage rapide de $MATLAB$, nous sommes capables de générer une émulation du micro contrôleur. Celui ci va s'occuper de récupérer les sorties mesurées du banc moteur, calculer la commande correspondantes et la générer sur l'entrée de commande du moteur. Cette partie va utiliser le protocole SIL décrit en début de chapitre.

4.4.1 Adaptation du modèle

Pour cette étude, nous avons réutilisé le bloc de simulation utilisé pour le MIL. Nous le connectons au bloc conçu pour envoyer et recevoir des signaux analogiques de la carte E/S. Ensuite, un *build* de la simulation complète est nécessaire pour ensuite lancer une émulation de la commande en temps réel. Vous trouverez le schéma $SIMULINK$ que nous avons utilisé en annexe, figure (7.6).

4.4.2 Test et étude de performances

Nous avons à notre disposition plusieurs prototypes déjà étudié en MIL. Cependant, nous n'utiliserons que les prototypes 1 et 2, qui disposent des meilleurs performances. Pour cette partie, nous souhaitons surtout analyser si les contraintes de temps réel sont respectés. Nous sommes dans une émulation, c'est à dire que chaque temps, entre la réception et l'émission est compté.

4.4.2.1 Étude du Prototype 1

Les résultats obtenus sur l'émulation de la commande du prototype 1 vont vous être présentés dans cette partie. Les observations des performances du système semblent, pour une simple analyse de cette réponse,

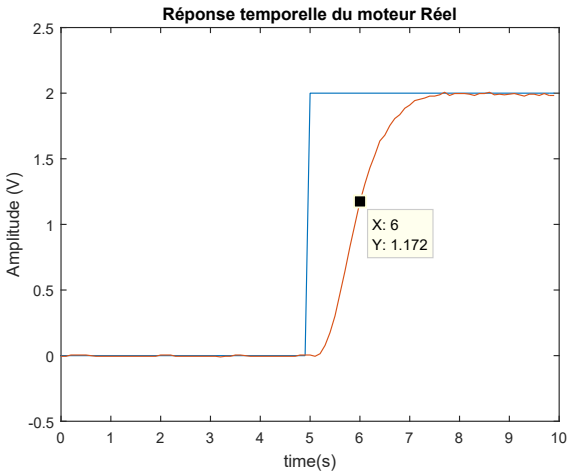


FIGURE 4.9 – Réponse teporelle $V_g(t)$, commande émulé et procédé réel

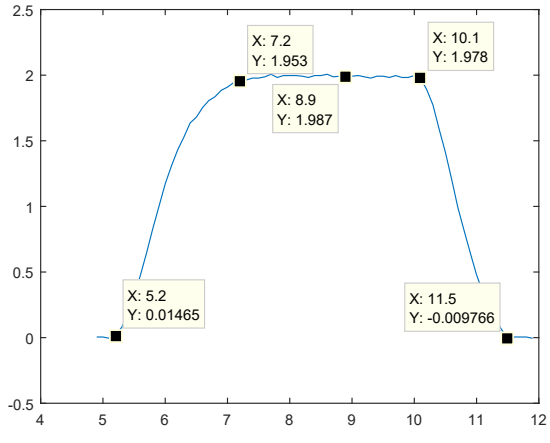


FIGURE 4.10 – Réponse teporelle $V_g(t)$, commande émulé et procédé réel, capture commentée

perturbées par le temps de réponse de l'un des deux éléments du test. Nous remarquons que $V_g(t)$ est beaucoup

plus lent qu'en simulation : la courbe de monté est décalé avec la consigne de manière significative. Les relevés donnent les résultats suivants : temps de montée $t_m = 2s$. Il est donc impératif de tester un nouveau prototype.

4.4.2.2 Étude du Prototype 2

Nous n'avons pas été en mesure de valider ce prototype, pour plusieurs raisons. Nous avons été retardé lors de la simulation de ce nouveau prototype, les multiples problèmes rencontrés lors de l'implémentation *SIMULINK* ne nous ont pas permis de faire des test sur le banc moteur.

4.4.3 Conclusion et Validation

Au cours de ce chapitre, nous avons pu aborder la mise en place des validations avec les protocoles utilisés pour y arriver. Grâce à ces outils, nous avons été capable de créer 3 prototypes de commande, un premier que nous allons abandonner du fait de ses mauvais résultats sur simulation avec un procédé non linéaire. Un deuxième prototype est quand à lui validé sur le modèle non linéaire mais doit être améliorer, i.e donner d'autres prototype, pour être validé sur émulation avec le moteur réel. Le troisième et dernier prototype est pour nous le plus compliqué mais aussi celui en qui possède le plus de chance de réussir : sa construction mathématique fait de ce prototype un bloc de calcul fort et il devrait sans doute converger vers la bonne solution à notre problème.

Nous allons maintenant passer au début de l'implémentation des prototypes sur des calculateurs électroniques. Nous devons pour cela analyser les contraintes liées à cette problématique, pour essayer d'écarter les situations qui pourrait amener nos prototypes à échouer dans la commande du moteur. Ces solutions amèneront la création de prototype qui seront des déroulements des deux prototype que nous avons gardé au bout de la SIL.

5 Commande temps discret

Voir annexe ?? pour les modèles simulink et les scripts d'affichages.
Voir annexe 7 pour le script de création de la commande à temps discret.

Afin d'implémenter la commande, nous allons adapter la commande de façon à ce qu'elle soit exécutable sur le micro-contrôleur. Ensuite, nous évaluerons cette transformation. Nous avons choisi d'implémenter la commande par retour d'état basé observateur car c'est celle que nous avons la plus aboutie compte tenu des spécifications.

5.1 Discrétisation de la commande

Nous allons à présent transformer la commande en temps continue en une forme qui permet l'implémentation sur un micro-contrôleur. Nous avons choisi de l'implémenter sous la forme d'une équation récurrente. Pour cela, dans un premier temps, nous avons transformer la commande (composée d'un observateur, d'un retour d'état et d'un pré-compensateur) en une forme espace d'états puis nous la discrétiserons. Ensuite, nous allons la transformer en une fonction de transfert pour finalement la transformer en équation récurrente.

5.1.1 Espace d'état à temps discret de la commande

La commande est de la forme suivante (figure 5.1) :

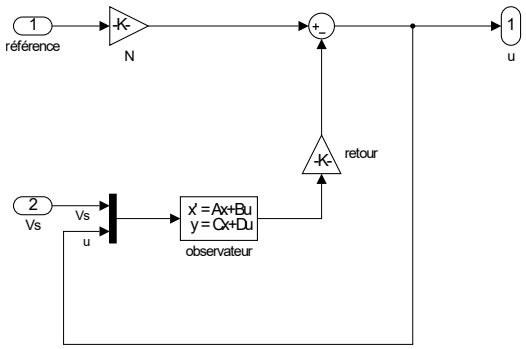


FIGURE 5.1 – Schéma de la commande par retour d'état à temps continue.

Voici, issue des équations ?? et 3.4, l'observateur simplifié (équation 5.1). Nous avons choisi de reconstruire l'observateur avec des valeurs propres désirées accélérée 3 fois car en simulation, cela réduit l'erreur de gain statique sur le système d'ordre 4 en boucle fermé.

$$\begin{cases} \dot{z}(t) = Fz(t) + GV_S(t) + Bu(t) \\ \hat{x}(t) = z(t) \end{cases} \tag{5.1}$$

Ainsi que le retour d'état :

$$u(t) = -K\hat{x}(t) + NV_{ref}(t) \tag{5.2}$$

Nous allons rassembler ces équations afin de créer un nouvel espace d'état :

$$\begin{cases} \dot{z}(t) = Fz(t) + GV_S(t) + Bu(t) \\ u(t) = -Kz(t) + NV_{ref}(t) \end{cases} \Leftrightarrow \begin{cases} \dot{z}(t) = (F - KB)z(t) + GV_S(t) + BNV_{ref}(t) \\ u(t) = -Kz(t) + NV_{ref}(t) \end{cases}$$

$$\Leftrightarrow \begin{cases} \dot{z}(t) = (F - KB)z(t) + \begin{bmatrix} BN & G \end{bmatrix} \begin{bmatrix} V_{ref}(t) \\ V_S(t) \end{bmatrix} \\ u(t) = -Kz(t) + \begin{bmatrix} N & 0 \end{bmatrix} \begin{bmatrix} V_{ref}(t) \\ V_S(t) \end{bmatrix} \end{cases} \tag{5.3}$$

Maintenant que nous avons exprimer la commande sous la forme d'un espace d'état, avec pour entrées $V_{ref}(t)$ la consigne et $V_S(t)$, la sortie de position mesurée, nous pouvons discrétiser la commande. Nous avons choisis de discrétiser la commande grâce à matlab, avec l'option *zoh* afin de discrétiser la commande avec un bloqueur d'ordre 0 :

$$B_O(p) = \frac{1 - e^{-pT_e}}{p} \tag{5.4}$$

Où $Te = 0.052$ secondes, c'est a dire 5.2 fois la fréquence maximale propre au moteur (voir chapitre 6. Cela donne la commande à temps discret suivante :

$$\begin{cases} z(z-1) = Az(z) + B \begin{bmatrix} V_{ref}(t) \\ V_S(t) \end{bmatrix} \\ u(z) = Cz(z) + D \begin{bmatrix} V_{ref}(t) \\ V_S(t) \end{bmatrix} \end{cases} \tag{5.5}$$

$$\Leftrightarrow \begin{cases} z(z-1) = \begin{bmatrix} -23.05 & 1 \\ -88.94 & -9 \end{bmatrix} z(z) + \begin{bmatrix} 0 & 20.74 \\ 85.71 & 80.05 \end{bmatrix} \begin{bmatrix} V_{ref}(t) \\ V_S(t) \end{bmatrix} \\ u(z) = \begin{bmatrix} 0 & -0.08899 \end{bmatrix} z(z) + \begin{bmatrix} 1.511 & 0 \end{bmatrix} \begin{bmatrix} V_{ref}(t) \\ V_S(t) \end{bmatrix} \end{cases}$$

5.1.2 Équation récurrente

Nous allons maintenant transformer l'espace d'état de l'équation 5.5 en fonctions de transferts, à l'aide de matlab :

$$\frac{u(z)}{V_{ref}(z)} = \frac{1.511z^2 - 1.551z + 0.3757}{z^2 - 0.8231z + 0.1889} \tag{5.6}$$

$$\frac{u(z)}{V_s(z)} = \frac{-0.1581z + 0.1581}{z^2 - 0.8231z + 0.1889} \tag{5.7}$$

Afin d'avoir une commande causale, nous avons passer les fonctions de transferts en z^{-1} :

$$\frac{u(z^{-1})}{V_{ref}(z^{-1})} = \frac{1.511 - 1.551z^{-1} + 0.3757z^{-2}}{1 - 0.8231z^{-1} + 0.1889z^{-2}} \tag{5.8}$$

$$\frac{u(z^{-1})}{V_s(z^{-1})} = \frac{-0.1581z^{-1} + 0.1581z^{-2}}{1 - 0.8231z^{-1} + 0.1889z^{-2}} \tag{5.9}$$

Nous avons ensuite, à partir des équations 5.8 et 5.9, créer une seule équation :

$$\begin{aligned} u(z^{-1})(1 - 0.8231z^{-1} + 0.1889z^{-2}) = {} & (1.511 - 1.551z^{-1} + 0.3757z^{-2})V_{ref}(z^{-1}) \\ & + (-0.1581z^{-1} + 0.1581z^{-2})V_s(z^{-1}) \end{aligned} \tag{5.10}$$

Nous avons maintenant une seule équation pour représenter la commande à temps discret.Il faut maintenant la transformée en équation récurrente, pour cela nous allons utiliser la propriété suivante :

$$f(z)z^n = f_{k+n} \tag{5.11}$$

Cela donne, une fois réorganiser de façon à isoler la sortie u_k et à partir de l'équation 5.10 :

$$\begin{aligned} u_k &= 0.8231u_{k-1} - 0.1889u_{k-2} \\ &\quad +1.511Vr_k - 1.551Vr_{k-1} + 0.3757Vr_{k-2} \\ &\quad -0.1581Vs_{k-1} + 0.1581Vs_{k-2} \end{aligned} \tag{5.12}$$

Où, par soucis de lisibilité, $Vs = V_s$ et $Vr = V_{ref}$. Le coût temporel de ce calcul est au minimum de $7 * 400ns = 2.8ms$.

5.2 Test de la commande en simulation

Afin de vérifier que la commande est bien fonctionnelle à temps discret, nous avons créer un modèle Simulink qui test la commande sous la forme d'une équation récurrente sur les modèles d'ordre 2, 3 et 4. Voici, figure 5.2, les réponses de ces asservissements à un signal rectangulaire d'amplitude 3 Volts.

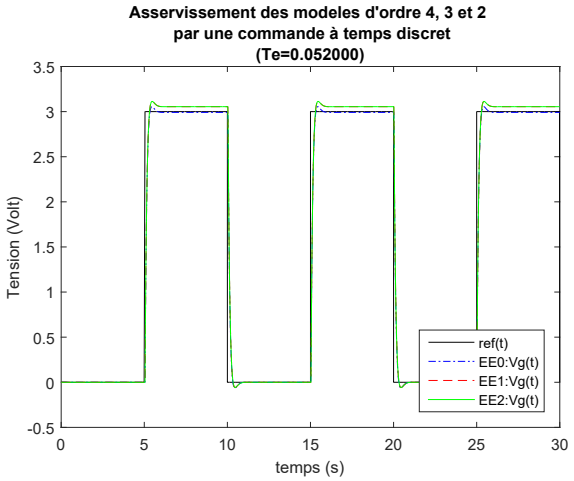


FIGURE 5.2 – Réponse des systèmes en boucle fermée asservis par équations récurrentes.

Nous pouvons voir que l'erreur statique et le temps de réponse sont important sur l'ordre 2 mais réduisent sur les ordres 3 et 4.

Maintenant que nous avons une équation implémentable sur micro-contrôleur, il faut créer un code le permettant, à partir de celui fourni.

6 Implémentation

Dans ce chapitre nous aborderons les problématiques lié à l'implémentation. Nous allons, dans un premier temps, évaluer les contraintes liés au support d'implémentation et dans un second temps les conversions que nous avons effectuer pour que l'entrée et la sortie de notre commande correspondent puissent être utilisé sur le micro-contrôleur. Finalement, nous expliquerons notre implémentation.

Voici, ci-dessous figure 6.1, le schéma général des différents éléments. On remarque qu'il y a 4 composants

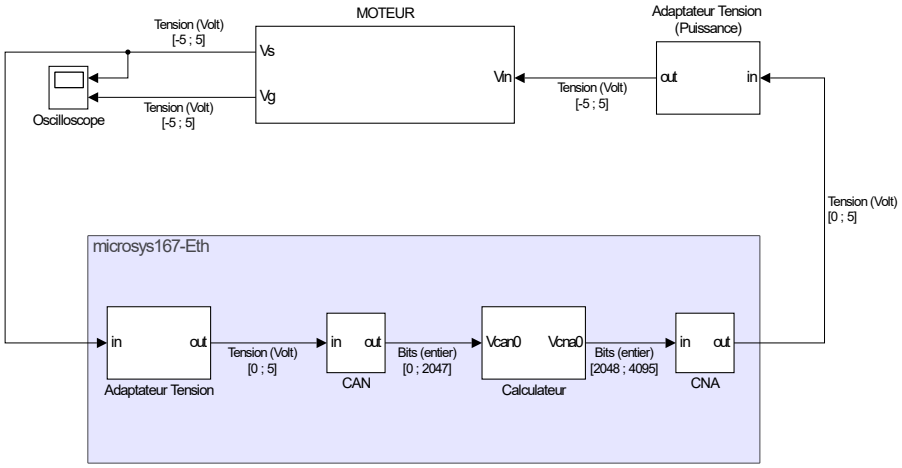


FIGURE 6.1 – Schéma des différents composants du montage.

et le détail, non exhaustif, des éléments utiles de la carte *microsys167-Eth*. Le bloc nommé moteur correspond au banc moteur présent en salle de TP. Le scope correspond à l'oscilloscope. Le bloc adaptateur de tension (Puissance) est la carte qui permet de passer de la tension de commande (0 à 5 Volt) vers une tension adaptée au moteur (-5 à 5 Volt).

6.1 Contraintes hardware

Nous allons, dans cette partie, nous consacrer à une étude des contraintes matérielles. Dans un premier temps nous verrons quels sont les spécificités du micro-contrôleur. Puis, nous verrons les convertisseurs analogique/numérique et numérique/analogique et nous conclurons sur le choix du micro-contrôleur.

6.1.1 Micro-contrôleur C167

Pour ce projet, nous disposons d'un micro-contrôleur *C167* fabriqué par Siemens. Il est dans une carte *microsys167-Eth* qui lui ajoute des interfaces, des fonctionnalités et de la mémoire. La carte cadence le micro-contrôleur à 20 MHz, ajoute un 1 Mo de RAM et 512 Ko de Flash-EPROM. Nous disposons aussi d'adaptateurs de tension ($[-5; 5]$ Volt vers $[0; 5]$ Volt). Le *C167* offre différentes fonctionnalités, dont les interruptions matérielles, les taches, les timers périodiques et un débogueur. Nous avons également des outils de développement permettant depuis un ordinateur, de créer un programme en *C*, le compiler pour le *C167*, l'envoyer sur celui-ci et si besoin de le déboguer et récupérer des valeurs sur un terminal. La fréquence de fonctionnement du *C167* est de 20 MHz et un cycle instruction faut 4 cycle CPU, donc la fréquence d'instruction est de 5 MHz. Le *C167* a également des entrées et des sorties numériques et analogiques. Nous allons maintenant vous parlez des problématiques de conversions.

6.1.2 CAN / CNA

6.1.2.1 CAN

Nous allons détailler deux types de conversions nécessaires à ce projet.
CAN : (Conversion Analogique Numérique.) Durant cette opération, le convertisseur échantillonne grâce à un bloqueur (discrétisation temporelle) puis quantifie (discrétisation de l'amplitude) le signal analogique. Il restitue un signal numérique après un temps de conversion t_{CAN}. Nous avons besoin de ce type de convertisseur pour la lecture des entrées dans notre cas, V_D. La qualité de notre conversion dépend de plusieurs éléments :

— Le nombre de bits de sortie (la sortie ne pouvant prendre que 2^{nbrBit} valeurs différentes). Nous avons des CAN 11 bits.

— La fréquence de conversion du signal analogique f_e par rapport à la fréquence utile maximale du signal a convertir f_{max}. Le théorème de Shannon préconise d'avoir au moins le rapport de l'équation 6.1, en pratique nous respecterons le rapport de l'équation 6.2. Cela évite le repliement.

$$f_e > 2 * f_{max} \tag{6.1}$$
$$f_e \approx 5 * f_{max} \tag{6.2}$$

Nous avons considéré que la fréquence utile maximale de la position d'un moteur à courant continue est d'environ $f_{max} = 100Hz$. Nous avons donc pris :

$$f_e = 520\text{Hz} \tag{6.3}$$
$$T_e = 52\text{ms} \tag{6.4}$$
$$\tag{6.5}$$

— La conversion doit être linéaire. En effet, il existe différentes erreurs possibles. Il peut y avoir un offset, une erreur de gainn une non-linéarité intégrale ou différentielle (voir figure 6.4).

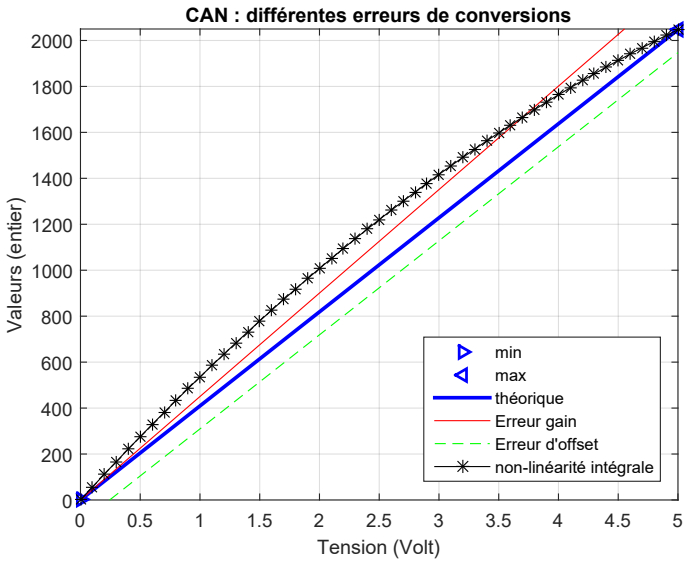


FIGURE 6.2 – Erreurs de conversions possibles sur le CAN

6.1.2.2 CNA

CNA : (Conversion Numérique Analogique.) Ici, le convertisseur transforme le signal numérique en un signal analogique après un temps de conversion t_{CNA} (équation 6.6). Nous avons besoin de ce type de convertisseur pour la génération de la sortie dans notre cas, V_s. La qualité de notre conversion dépend de plusieurs éléments similaires au CAN. L'entrée du CNA est un entier codé sur 12 bits et il génère une tension comprise entre -5 et 5 Volts.

$$t_{CAN} = 14 * t_{cc} + 2 * t_{sc} + 4 * TLC \tag{6.6}$$
$$= 9,7\mu s \tag{6.7}$$

Par manque de temps, nous n'avons pas pu mettre en place un protocole complet d'étude des erreurs de conversions, nous avons uniquement émis des valeurs avec le CNA et les avons lu avec le CAN afin que les valeurs mesurées et générées correspondent à peu prés à celles désirées. Nous avons fait ceci pour 1000 valeurs. Nous avons réaliser ce test pour les deux micro-contrôleurs et cela donne des résultats différents. La figure 6.3 est le résultat de l'analyse du micro-contrôleur de gauche en salle de TP. Nous voyons que la conversion est assez mauvaise, d'autres tests ont mis en évidence que le problème vient du CAN : de multiples lectures d'une tension constante donnent des résultats avec une variation importante, trop pour être du bruit numérique ou électromagnétique. Nous avons ré effectué le même test sur le second micro-contrôleur (celui de droite) et, figure 6.4, on remarque qu'il y a deux saturations : avant $E1$ et après $E2$. Entre les deux, la conversion est à peu près constante. Le second C167 est donc plus fiable en terme de lecture et d'écriture de tensions.

Cette approche présente l'avantage d'être rapide mais elle est néanmoins peu précise. En effet, les erreurs du CAN peuvent être masqués par celles du CNA et inversement. Elle ne permet pas de savoir d'où vient le problème, à moins d'effectuer d'autres expériences sur le CAN et/ou le CNA.

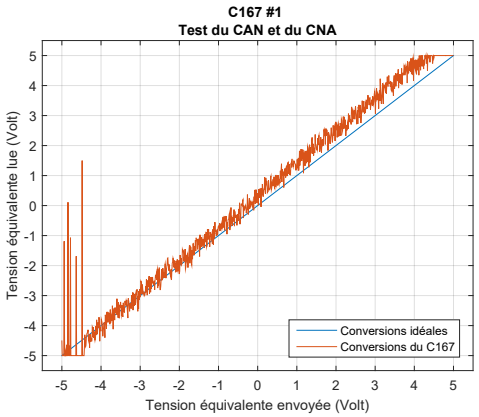


FIGURE 6.3 – Erreurs de conversions entre le CAN et le CNA (pour le C167 n°1)

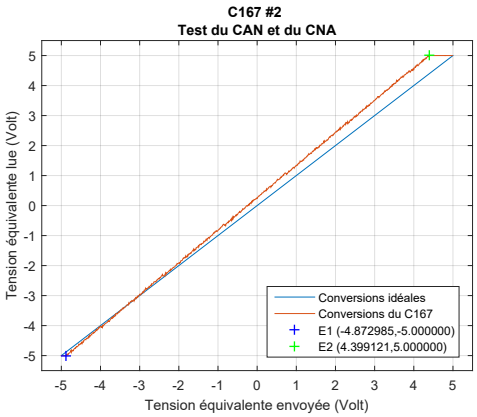


FIGURE 6.4 – Erreurs de conversions entre le CAN et le CNA (pour le C167 n°2)

Un protocole de test complet aurait été :

- Pour le CAN :
  - Générer, à partir d'un générateur de basse tension, un signal triangle de 0 à 5 Volt de fréquence faible.
  - Lire toutes les valeurs et les faire afficher sur un terminal par le micro-contrôleur.
  - Les comparer (à l'aide d'un tableur ou de Matlab) et vérifier la linéarité de la conversion.
  - À partir des ces résultats, créer une fonction qui corrige les erreurs, si possible.
- Pour le CNA :
  - Générer, à partir du micro-contrôleur un signal triangle de -5 à 5 Volt de fréquence faible.
  - Récupérer les valeurs à partir d'un CAN déjà corrigé (carte d'acquisition Matlab, ou le CAN précédemment si la correction donne de très bons résultats).
  - Les comparer (à l'aide d'un tableur ou de Matlab) et vérifier la linéarité de la conversion.
  - À partir des ces résultats, créer une fonction qui corrige les erreurs, si possible.

6.2 Transformation CAN/CNA

6.2.1 CAN

Afin de réalisé notre commande, nous avons dût transformer la valeur entière lue pour V_S par le CAN en un équivalent de la tension de type nombre à virgule flottante. Figure 6.5, nous pouvons observer les valeurs que peut convertir le CAN et les valeurs dont nous avons besoins. Nous n'utilisons ici que la moitié de la valeur lue par le convertisseur car nous n'avons besoin de convertir que des valeurs comprises entre 0 et 5 Volt. Ici, la plage de conversion est donc à moitié utilisée. Dans un premier temps, nous avons décaler la valeur lue (avec un et bit a bit) afin d'avoir la valeur sur $[0, 1023]$, puis nous avons réalisé la conversion en tension avec l'équation 6.8.

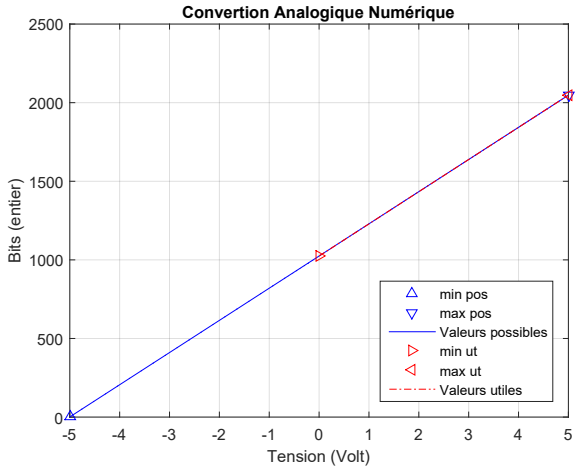


FIGURE 6.5 – Valeurs possibles et possibles du CAN

$$\begin{cases} V_{CANr} &= V_{CAN} \&_{bb} 1023 \\ V_{volt} &= \frac{5}{2^{10}-1} V_{CANr} \end{cases} \qquad (6.8)$$
$$\Rightarrow V_{volt} = \frac{5}{2^{10}-1}(V_{CAN} \&_{bb} 1023)$$

6.2.2 CNA

La génération de la sortie V_M nécessite aussi une conversion du CNA depuis la tension calculée par la commande (nombre à virgule flottante) vers un entier compris entre -5 et 5 Volts. Néanmoins, la commande calcule une valeur comprise entre -5 et 5 Volts, mais la carte de puissance branchée sur la sortie du CNA nécessite en entrée une tension comprise entre 0 et 5 Volt. Il faut donc, dans un premier temps, redresser la valeur calculée V_{com} par la sortie sur une intervalle $[0;5]$ V_{redr}, puis le convertir en entier V_{CNA}. Pour cela nous utilisons les équations suivantes :

$$\begin{cases} V_{red} &= \frac{1}{2} V_{com} + 2,5 \\ V_{CNA} &= \frac{2^{11}-1}{5} V_{red} + 2048 \end{cases} \qquad (6.9)$$
$$\Rightarrow V_{CNA} = \lfloor 204.7 V_{com} + 3071.5 \rfloor$$

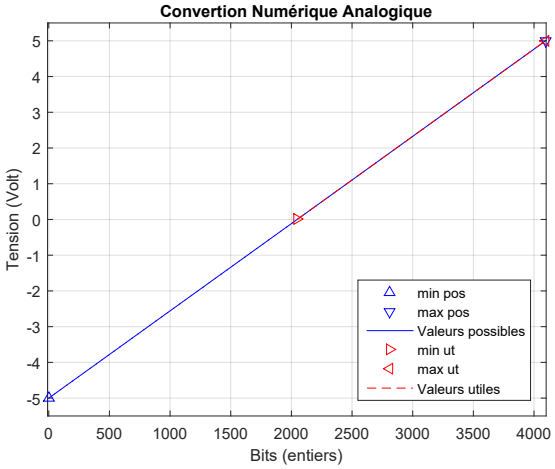


FIGURE 6.6 – Valeurs possibles et possibles du CNA

6.3 Implémentation sur micro-contrôleur

Nous souhaitons dans cette partie vous présenter comment nous avons aborder l'implémentation des commandes sur le micro-contrôleur. Nous aborderons dans un premier temps une revue des tâches que nous avons utilisées.

6.3.1 Description des taches

Génération échelon Une première tâche sera dédié à la génération d'un échelon. Nous souhaitons observer des changements de front de la consigne, c'est pourquoi il est intéressant de faire évoluer cette consigne périodiquement. Dans cette tâche, qui déroule du squelette obtenu sur les fichiers sources, nous allons avoir le déroulement suivant :

État 1 Valeur de la consigne mise à l'état haut. Variable d'état = 2

État 2 Valeur de la consigne mise à l'état bas. Variable d'état = 1

La fréquence d'appel de cette tâche n'a pas de grand impact sur les contraintes temps réel. Les timers ont déjà été programmé sur une fréquence de $f_{ech} = \frac{3.36}{2} Hz$, nous garderons ce résultat dans le reste de l'étude.

6.3.1.0.1 Acquisition des mesures Cette tâche beaucoup plus principale que la précédente doit permettre de récupérer la valeur lue sur le CAN, calculer la commande associée et la transmettre au CNA. Ce déroulement a été décrit dans le schéma 6.1. Il est important de faire attention au conversion obtenu dans les équations 6.9 et 6.8.

6.3.2 Implémentation

Vous trouverez le code implémenté en langage C des tâches dans l'annexe 7. Nous avons aussi inclus un fichier de génération de paramètres via un modèle discret de *MATLAB*. Ce fichier génère un *.txt* en fonction des paramètres de l'équation récurrente du prototype de commande.

6.3.3 Validation et correction

Nous avons obtenu des problèmes pour le prototype qui a été discrétisé dans la partie 5 et qui est contenu dans le programme. Il semble que certaines problématiques nous ait échappé, notamment les notions de correction des CAN CNA évoquée dans 6.1.2. Pour évaluer si le problème venait de la conversion, nous avons tout de même été capable de générer un échelon de tension qui vous est présenté dans la figure ci dessous.

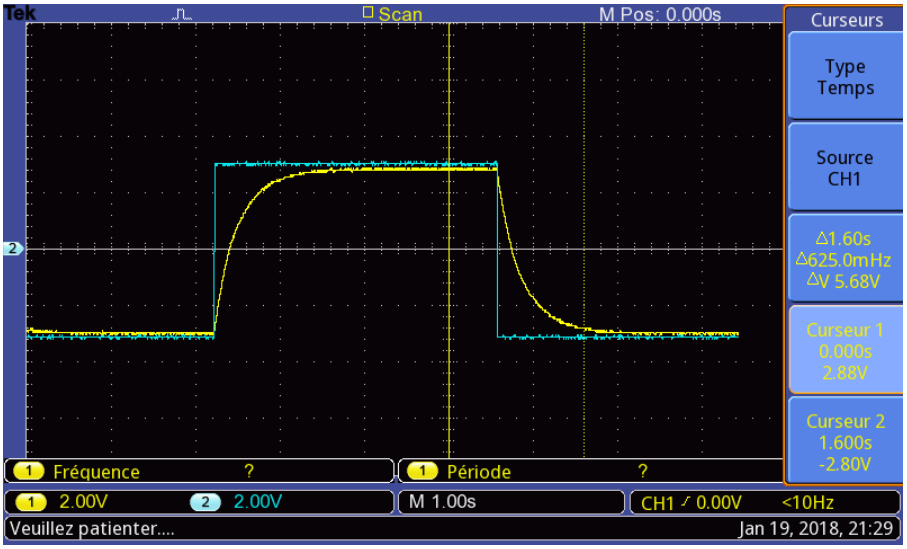


FIGURE 6.7 – Réponse à un échelon +2V, -2V

7 | Bilan

Nous venons de vous présenter l'ensemble de nos travaux sur la conception de commande temps réel avec : la modélisation du procédé et son analyse, une synthèse de commande avec sa validation ainsi que sa discrétisation, en terminant par son implémentation sur un code en langage C.

Pour chacune de ces parties, nos compétences acquises au cours de notre cursus ont été utiles et nous avons relevé un grand intérêt dans la mise en relation entre ces différentes connaissances. Nous avons pu faire un lien entre l'automatique linéaire et l'implémentation sur micro-contrôleur, en passant par la discrétisation et l'analyse de prototype avec *MATLAB*. Nous avons vu aussi l'importance lié à une validation des résultats étape après étape, pour ainsi effectuer une mise en œuvre un peu plus réfléchie que durant nos années précédente.

Cependant, nous avons rencontré quelques problèmes qui sont, il nous semble, bon de faire ressortir pour une évolution personnelles. Nous avons été en difficulté quand à la sélection des points précis qu'il fallait aborder. En effet, nous ne pouvions pas utiliser l'intégralité des compétences, un tri a été nécessaire et nous avons été très souvent indécis quand à l'utilisation ou non d'un point précis de la compétence.

En ouverture de ces travaux, nous remarquons qu'il est important d'appréhender de manière plus objective les compétences qui seront utilisé. Nous savons aussi maintenant à quel point il est essentiel de hiérarchiser la conception, en passant par chaque étape de validation qui ont été choisi avec précautions et toute connaissance du problème posé.

Annexes

Annexe 1 - Scripts Matlab

Modèles et analyses

```
clear
close all
MoteurScript
%% Modele Niveau 0 - Ordre 4 - EE
%%% Modele espace d'etat LINEAIRE
% etat    = [ i1 ; i2 ; theta ; omega ]
% entree  = [ Vm ]
% sorties = [ Vg ; Vs]

% Matrices EE

EE0.a   =[-R/L,     0,              0,    -Ke/L;
          0,        -(R+Rchn)/L,    0,    -Ke/L;
          0,        0,              0,    1;
          Kc/J2,    Kc/J2,          0,    -mu/J2];

EE0.b   =[1/L;
          0;
          0;
          0];

EE0.c   =[0, 0, 0,       Kg;
          0, 0, Kr*Ks,  0 ];

EE0.d   =[0;
          0];
% EE
EE0.ee= ss(EE0.a,EE0.b,EE0.c,EE0.d);
% Valeurs propres
EE0.vp = eig(EE0.ee);

% gain statique
EE0.gain = dcgain(EE0.ee(1));
%% Modele Niveau 1 - Ordre 3 - EE
%%% Modele espace d'etat LINEAIRE
% etat    = [ i1 ; omega ; theta ]
% entree  = [ Vm ]
% sorties = [ Vg ; Vs]

% Matrices EE
EE1.a = EE0.a([1,3,4],[1,3,4]);
EE1.b = EE0.b([1, 3, 4]);
EE1.c = EE0.c(:,[1, 3, 4]);
EE1.d = EE0.d;

% Espace Etat
```

```
47 EE1.ee= ss(EE1.a,EE1.b,EE1.c,EE1.d);
48 % Valeurs propres
49 EE1.vp = eig(EE1.ee);
50
51 % gain statique
52 EE1.gain = dcgain(EE1.ee(1));
53 %% Modele Niveau 2 - Ordre 2 - EE
54 %%% Modele espace d'etat LINEAIRE
55 % etat    = [ theta ; omega ]
56 % entree  = [ Vm ]
57 % sorties = [ Vg ; Vs]
58
59 % Matrices EE
60 % EE2.a = [ -(Ks*Ke)/(J2*R) , 0 ;
61 %           1                , 0 ]; % ca marche pas...
62 EE2.a = [  0  ,   1 ;
63            0  ,   -(Kc*Ke)/(R*J2)-(mu/J2)];
64 EE2.b = [ 0          ;
65           Kc/(J2*R) ];
66
67 EE2.c = EE1.c(:,[2, 3]);
68 %EE2.c =[];
69 EE2.d = EE1.d;
70
71 % EE
72 EE2.ee= ss(EE2.a,EE2.b,EE2.c,EE2.d);
73 % Valeurs propres
74
75 EE2.vp = eig(EE2.ee);
76
77 % gain statique
78 EE2.gain = dcgain(EE2.ee(1));
79
80 % Commandabilitee
81 Controlabilite = ctrb(EE0.a,EE0.b);
82 disp('Controlable ?')
83 disp(rank(Controlabilite) == size(EE0.a,1))
84
85 % Observabilite
86
87 [ABAR,BBAR,CBAR,T,K] = obsvf(EE0.a,EE0.b,EE0.c)
88
89 % etude de performance temporelle
90 EE0.stepChar = stepinfo(EE0.ee(1),'SettingTime',0.05)
91 EE1.stepChar = stepinfo(EE1.ee(1),'SettingTime',0.05)
92 EE2.stepChar = stepinfo(EE2.ee(1),'SettingTime',0.05)
```

Observateur et Asservissement

```
1
2 %% Creation de notre observateur
3 % reconstruction de tous les etats :
4 espaceEtat1
5
6 %% Cahier des charges
7 vp_desire = [-4 ; - 5];
8 %% observateur
9 obsver.H = EE2.ee.b;
10 obsver.M = eye(size(EE2.ee.a));
11 obsver.vp =  vp_desire; %*3
```

```
obsver.G = place(EE2.ee.a', EE2.ee.c(2,:)', obsver.vp)'; % Ne pas utiliser
    acker
%%
obsver.F = EE2.ee.a - obsver.G*EE2.ee.c(2,:);
           %   a           b                   c       D
obsver.ee = ss(obsver.F, [obsver.G obsver.H], obsver.M, 0);

%% commande

K = place(EE2.ee.a, EE2.ee.b, vp_desire);
K = [0 K(2)];
EE2_bf.a = [EE2.ee.a-EE2.ee.b*K     -EE2.ee.b*K;
             [0 0; 0 0]              obsver.F];

EE2_bf.b = [EE2.ee.b; 0;0];

EE2_bf.c = [EE2.ee.c [0 0;0 0]];

EE2_bf.ee = ss(EE2_bf.a, EE2_bf.b, EE2_bf.c, EE2.ee.d);

EE2_bf.gain = dcgain(EE2_bf.ee(1));
%   N = 1/EE2_bf.gain; %Modif En salle
%%  Analyse du retour d'etat base observateur
% Pour EE1 :
%Changement de base de EE1
P = [0     0     1;
     1     0     0;
     0     1     0];
EE1_c.a = inv(P)*EE1.ee.a*P;
EE1_c.b = inv(P)*EE1.ee.b;
EE1_c.c = EE1.ee.c*P;

EE1_c.ee = ss(EE1_c.a,EE1_c.b,EE1_c.c,EE1.ee.d);

% Partitionnement de l'EE
EE1.A1 = EE1_c.ee.a(1:2,1:2);
EE1.A2 = EE1_c.ee.a(1:2,3);
EE1.A3 = EE1_c.ee.a(3,1:2);
EE1.A4 = EE1_c.ee.a(3,3);

EE1.B1 = EE1_c.ee.b(1:2);
EE1.B2 = EE1_c.ee.b(3);

EE1.C = [EE1_c.ee.c];


% Construction des matrices
EE1_obsver.A = [EE1.A1      EE1.A2     [0 0;0 0]   ;
                EE1.A3      EE1.A4      [0 0 ] ;
                [0 0;0 0]   -EE1.A2       obsver.F];
EE1_obsver.B = [EE1.B1; EE1.B2; [0;0]];
EE1_obsver.C = [EE1.C [0 0;0 0];
                    [0 0 0 0 1]];   % Pour afffiche de l'erreur
                                    % de reconstrcution de la

% Espace d'etat
EE1_obsver.ee = ss(EE1_obsver.A ,EE1_obsver.B, EE1_obsver.C, [0;0;0]);


% gain statique
EE1_obsver.gain = dcgain(EE1_obsver.ee(1));
```

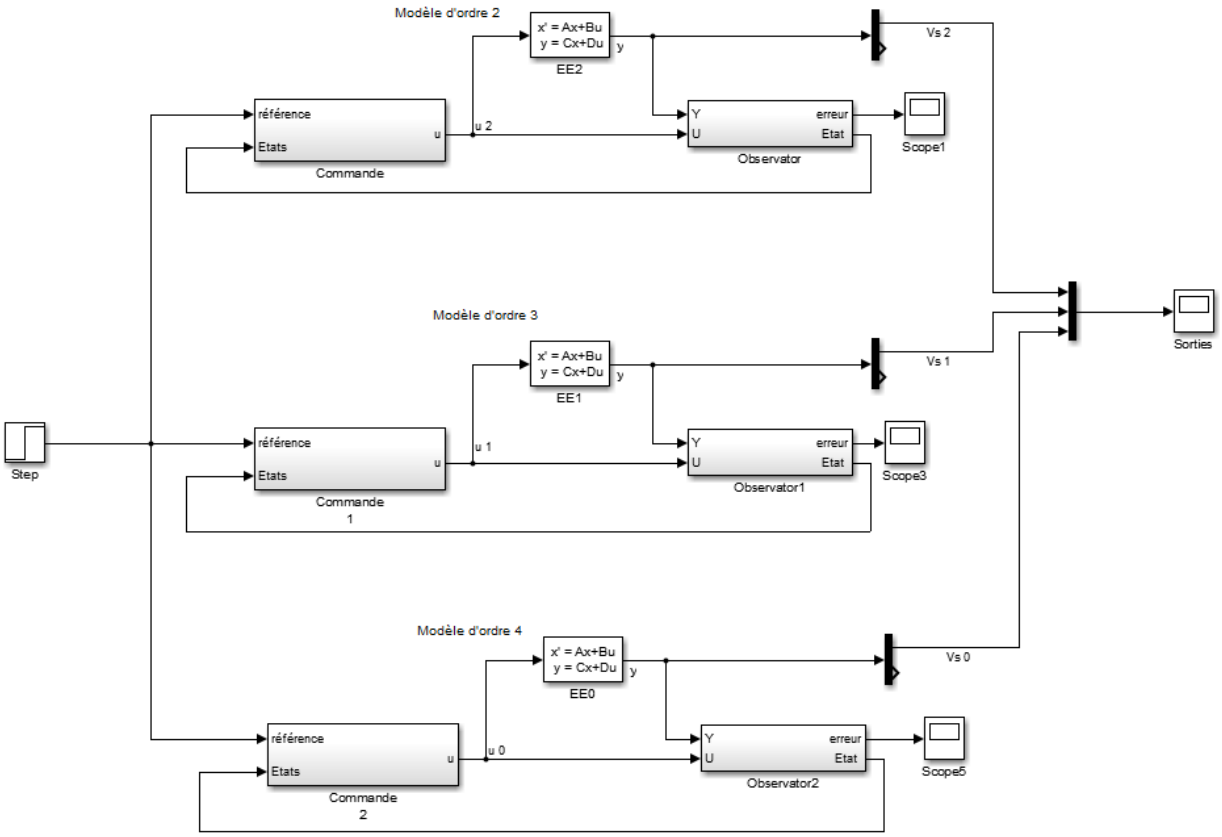
```
72 % comparaison de gain statique
73
74 err_gain_stat=EE2.gain - EE1_obsver.gain;
75
76 % Analyse pour EE0
77 % Changement de base [theta, omega, i1, i2]
78 P_0 = [0 0 1 0; 0 0 0 1; 1 0 0 0; 0 1 0 0];
79
80 EE0_c.a = inv(P_0)*EE0.ee.a*P_0;
81 EE0_c.b = inv(P_0)*EE0.ee.b;
82 EE0_c.c = EE0.ee.c*P_0;
83
84 EE0_c.ee = ss(EE0_c.a,EE0_c.b,EE0_c.c, EE0.d);
85 % Partitionnement de l'EE
86 EE0.A1 = EE0_c.ee.a(1:2,1:2);
87 EE0.A2 = EE0_c.ee.a(1:2,3:4);
88 EE0.A3 = EE0_c.ee.a(3:4,1:2);
89 EE0.A4 = EE0_c.ee.a(3:4,3:4);
90
91 EE0.B1 = EE0_c.ee.b(1:2);
92 EE0.B2 = EE0_c.ee.b(3:4);
93
94 EE0.C = [EE0_c.ee.c];
95
96
97 % Construction des matrices
98 EE0_obsver.A = [EE0.A1      EE0.A2     [0 0;0 0]   ;
99                 EE0.A3      EE0.A4      [0 0;0 0 ] ;
100                [0 0;0 0]   -EE0.A2       obsver.F];
101 EE0_obsver.B = [EE0.B1; EE0.B2; [0;0]];
102 EE0_obsver.C = [EE0.C [0 0;0 0]];   % Pour afffiche de l'erreur
103                                     % de reconstrcution de la
104
105 EE0_obsver.ee = ss(EE0_obsver.A, EE0_obsver.B, EE0_obsver.C, EE0_c.ee.d);
106
107 % Analyse du trasfert de epsilon
108 EE0_obsver.vp = eig(EE0_obsver.ee);
109 %%
110
111
112 %% Calcul du systeme en boucle ferme base observateur de EE1
113 %       cf Cours de M. GOUAISBAULT
114 % Etat EE0 : x = [ i1    ; i2    ; theta ; omega ]
115 % Etat EE1 : x = [ i1    ; theta ; omega ]
116 % Etat EE2 : x = [ theta ; omega ]
117
118 % EE1
119
120 EE1_bf.a = [EE1.A1-EE1.B1*K      EE1.A2     -EE1.B1*K   ;
121             EE1.A3-EE1.B2*K      EE1.A4     -EE1.B2*K;
122                 [0 0;0 0]       -EE1.A2       obsver.F];
123 EE1_bf.b = EE1_obsver.B;
124 EE1_bf.c = EE1_obsver.C;
125
126 EE1_bf.ee = ss(EE1_bf.a, EE1_bf.b, EE1_bf.c, EE1_obsver.ee.d);
127 EE1_bf.vp = eig(EE1_bf.ee);
128
129 % EE0
130
131
132 EE0_bf.a = [EE0.A1-EE0.B1*K      EE0.A2     -EE0.B1*K   ;
```

```
133             EE0.A3-EE0.B2*K      EE0.A4     -EE0.B2*K;
134                 [0 0;0 0]       -EE0.A2       obsver.F];
135 EE0_bf.b = EE0_obsver.B;
136 EE0_bf.c = EE0_obsver.C;
137
138 EE0_bf.ee = ss(EE0_bf.a, EE0_bf.b, EE0_bf.c, EE0_obsver.ee.d);
139 EE0_bf.vp = eig(EE0_bf.ee);
140 EE0_bf.gain = dcgain(EE0_bf.ee(1));
141
142 N = 1/(EE2_bf.gain);
143 %% Retour Integral :
144 %   Nouvel espace d'etat avec x = [x xi] = [theta; omega; (y(t)-yref(t)]
145 %
146 EEIntegral.a = [EE2.ee.a [0 ; 0];EE2.ee.c(2,:) 0];
147 EEIntegral.b = [EE2.ee.b ; 0];
148 EEIntegral.c = [EE2.ee.c [0;0]];
149 EEIntegral.d = EE2.ee.d;
150 % Nouvel gain de placement de pole : K = [Kp Ki]
151 % Kp --> poles; Ki --> integrateur
152 EEIntegral.K = place(EEIntegral.a, EEIntegral.b, [3*vp_desire ;-10]);
153
154
155
156 %% Bloc de commande temps discret
157 Te = 0.026;
158 %% input :
159 CommTD.a = [obsver.F-EE2.ee.b*K];
160 CommTD.b = [EE2.ee.b*N obsver.G];
161 CommTD.c = [-K];
162 CommTD.d = [N 0];
163 CommTD.ee = ss(CommTD.a, CommTD.b, CommTD.c, CommTD.d);
164 CommTD.zz = c2d(CommTD.ee, Te, 'zoh');
165 CommTD.ft=tf(CommTD.zz);
166 ftRef=CommTD.ft(1);
167 ftVs=CommTD.ft(2);
168 %%
169 ftTD=CommTD.ft(1)+CommTD.ft(2);
170
171 %% save in file
172 numY =CommTD.ft(1,1).num{1,1};
173 numU =CommTD.ft(1,2).num{1,1};
174 denumY = CommTD.ft(1,1).den{1,1};
175
176 File        = fopen('Parametres_Commande.txt','w+');
177
178 fprintf(File,'\t Parametres de la commande en temps discret \n\n');
179 %ny 1 2 3
180 fprintf(File,'static float my0 =  %f;\n',numY(1));
181 fprintf(File,'static float my1 =  %f;\n',numY(2));
182 fprintf(File,'static float my2 =  %f;\n\n',numY(3));
183 %d 1 2
184 fprintf(File,'static float d1 =  %f;\n',denumY(2));
185 fprintf(File,'static float d2 =  %f;\n\n',denumY(3));
186 % nu 1 2 3
187 fprintf(File,'static float mref0 =  %f;\n',numU(1));
188 fprintf(File,'static float mref1 =  %f;\n',numU(2));
189 fprintf(File,'static float mref2 =  %f;\n\n',numU(3));
190
191 fclose(File);
192
193 %% Bloc de commande  par retour d'etat avec effet integral
```

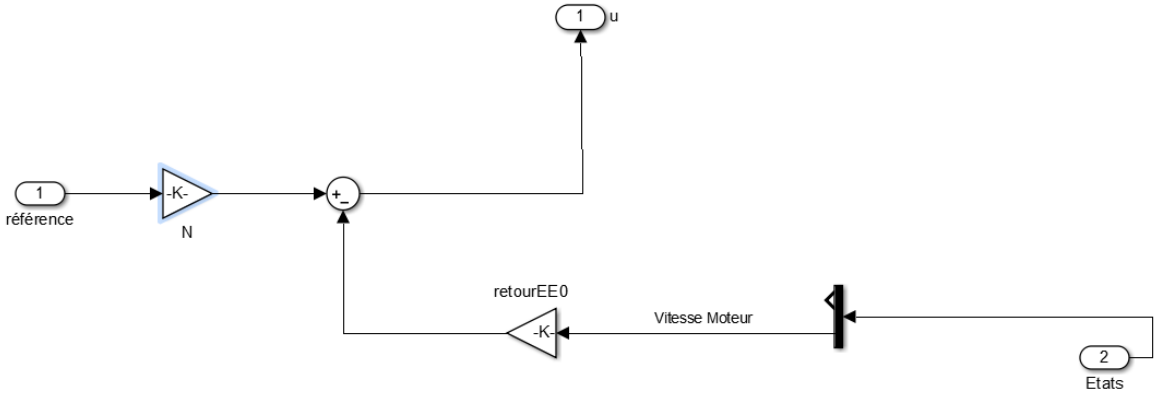
```
194 % Etat = [x_reconstruit ; xi]
195 % Sortie = u_commande
196 % Entree = [y_ref ; y]
197 CommTD_integral.a = [obsver.F-EE2.ee.b*[0 EEIntegral.K(2)]
       EE2.ee.b*EEIntegral.K(3); [-EE2.ee.c(1,:) 0]];
198 CommTD_integral.b = [[0;0] obsver.G ; 1 0];
199 CommTD_integral.c = [-[0 EEIntegral.K(2)] EEIntegral.K(3)];
200 CommTD_integral.d = [0 0];
201 CommTD_integral.ee = ss(CommTD_integral.a, CommTD_integral.b,
       CommTD_integral.c, CommTD_integral.d);
202 % Discretisation
203 CommTD_integral.td = c2d(CommTD_integral.ee,Te, 'zoh');
204 %step(CommTD_integral.ee);
205 CommTD_integral.ft=tf(CommTD_integral.td);
206 ftRef=CommTD_integral.ft(1);
207 ftVs=CommTD_integral.ft(2);
208 numY =CommTD_integral.ft(1,1).num{1,1};
209 numU =CommTD_integral.ft(1,2).num{1,1};
210 denumY = CommTD_integral.ft(1,1).den{1,1};
211 %% save in file
212
213 File         = fopen('Parametres_Commande_int.txt','w+');
214
215 fprintf(File,'\t Parametres de la commande en temps discret \n\n');
216 %ny 1 2 3
217 fprintf(File,'static float my0 =  %f;\n',numY(1));
218 fprintf(File,'static float my1 =  %f;\n',numY(2));
219 fprintf(File,'static float my2 =  %f;\n\n',numY(3));
220 fprintf(File,'static float my3 =  %f;\n\n',numY(4));
221 %d 1 2
222 fprintf(File,'static float d1 =  %f;\n\n',denumY(2));
223 fprintf(File,'static float d2 =  %f;\n\n',denumY(3));
224 fprintf(File,'static float d3 =  %f;\n\n',denumY(4));
225
226 % nu 1 2 3
227 fprintf(File,'static float mu0 =  %f;\n',numU(1));
228 fprintf(File,'static float mu1 =  %f;\n',numU(2));
229 fprintf(File,'static float mu2 =  %f;\n\n',numU(3));
230 fprintf(File,'static float mu3 =  %f;\n\n',numU(4));
231
232 fclose(File);
```

Annexe 2 - Modèles SIMULINK

Modèle Global



Sous-système de commande



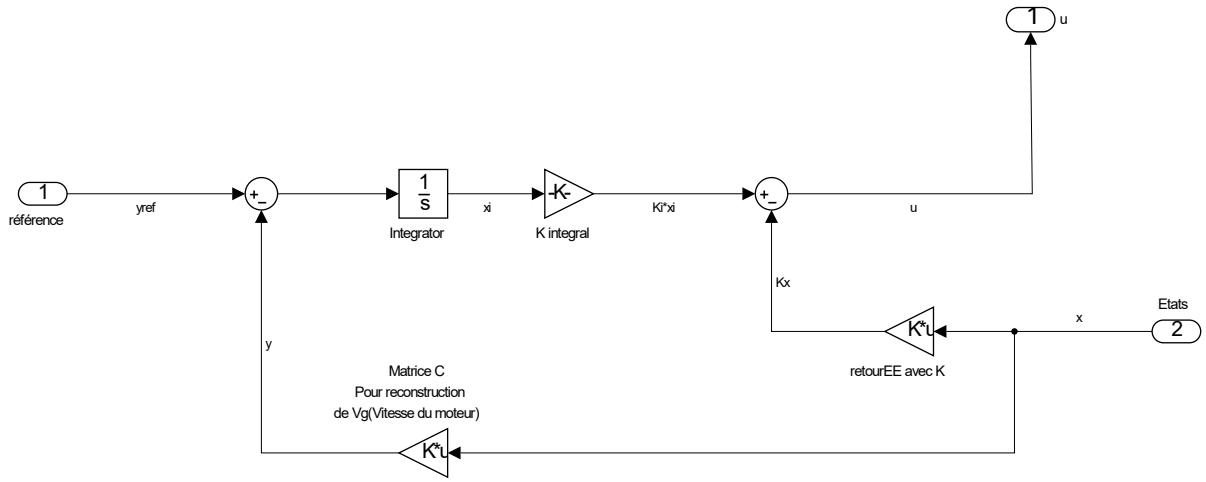
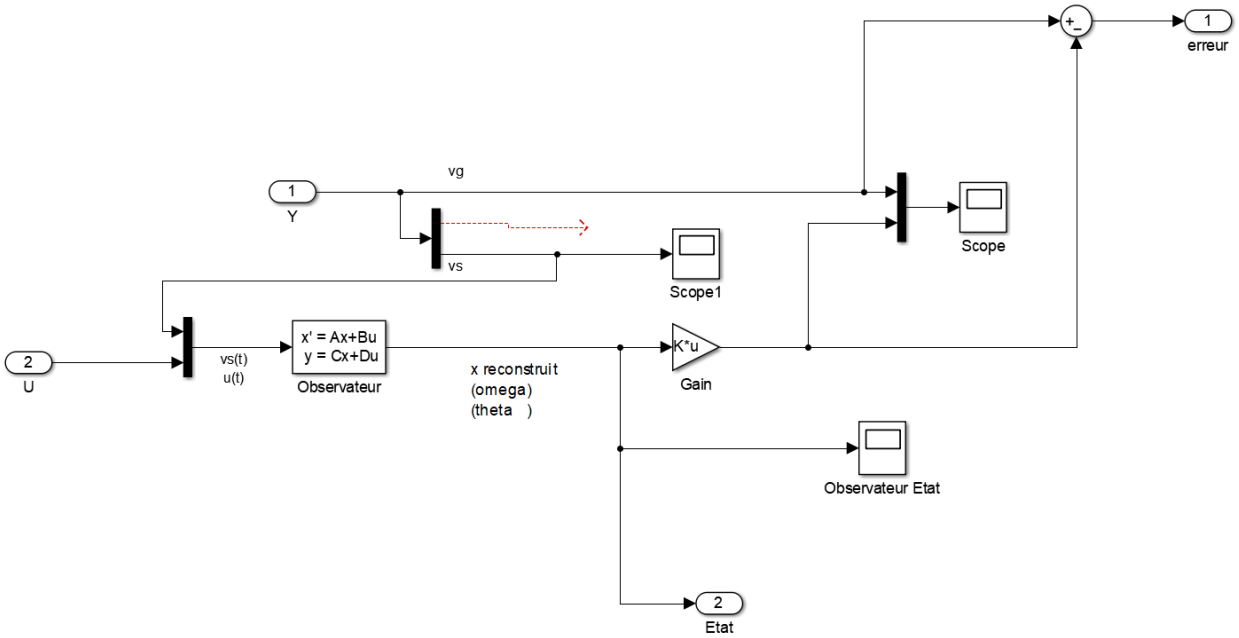


FIGURE 7.1 – Sous système de commande avec action Intégrale

Sous-système de commande avec action intégrale

Sous-système d'observateur



Annexe 3 - Simulation & modèles

Modèles Non linéaire

Schéma *SIMULNK*

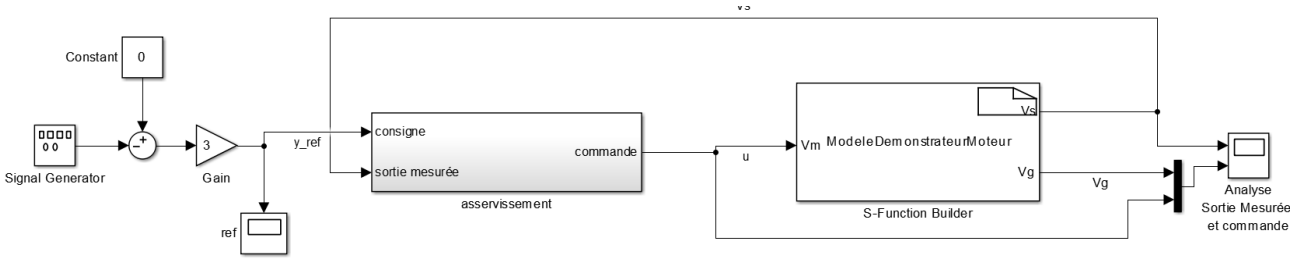


FIGURE 7.2 – Schéma *SIMULINK* complet de l'asservissement du modèle du moteur Non linéaire

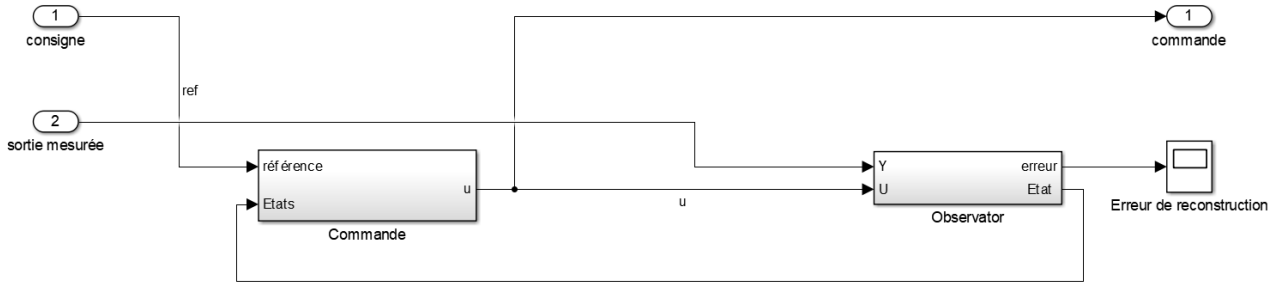


FIGURE 7.3 – Schéma *SIMULINK* du *sub-system* de commande

Réponses temporelles

Émulation Moteur réel

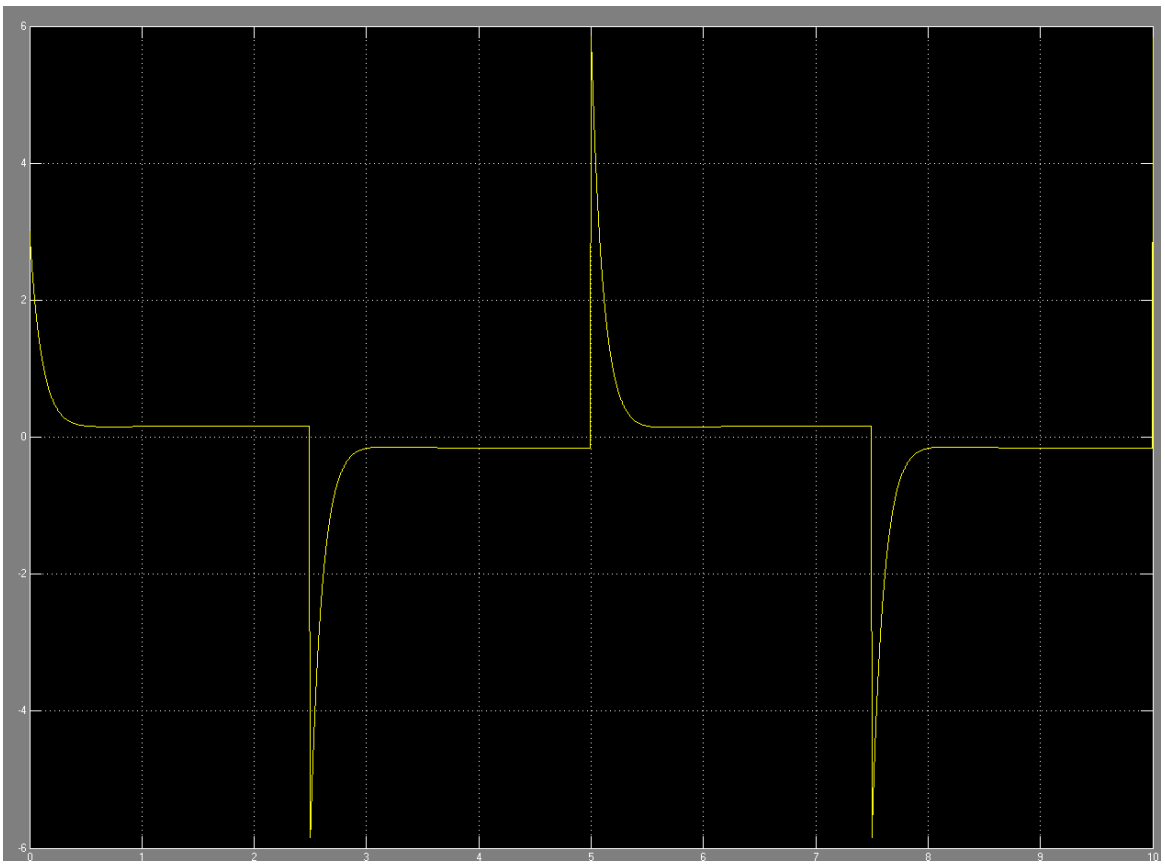


FIGURE 7.4 – Mesure de simulation de l'erreur entre la référence et la sortie V_s du modèle Non linéaire

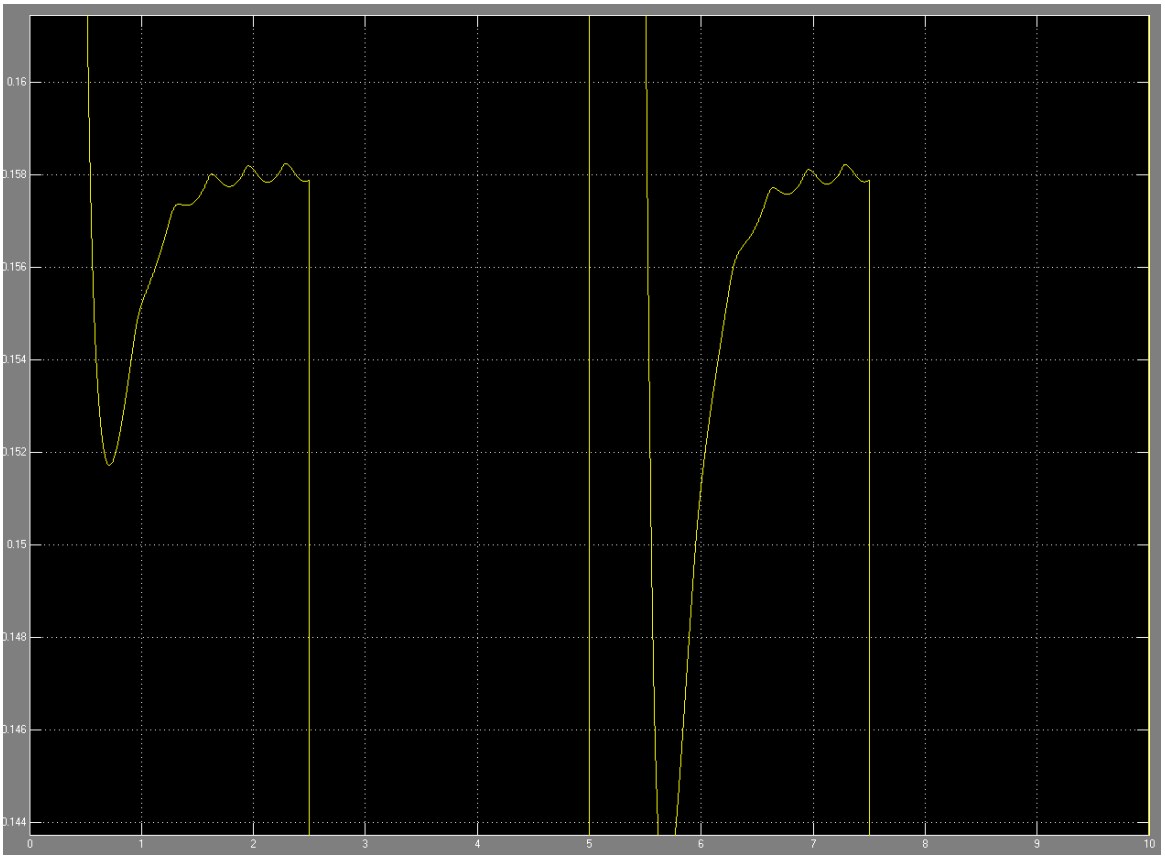


FIGURE 7.5 – Mesure de simulation de l'erreur ϵ du modèle Non linéaire centré sur l'axe des ordonnées en $[0.144; 0.16]$.

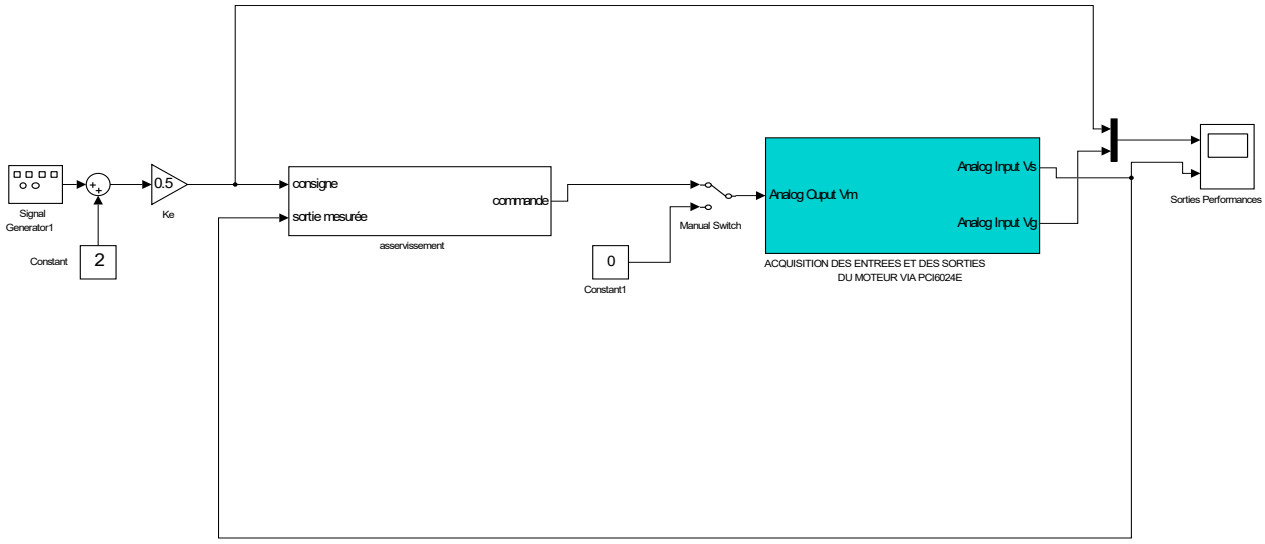


FIGURE 7.6 – Schéma des blocs *Simulink* de l'émulation de la commande sur moteur réel

Annexe 4 - Commande à temps discret

Modèles *SIMULNK*

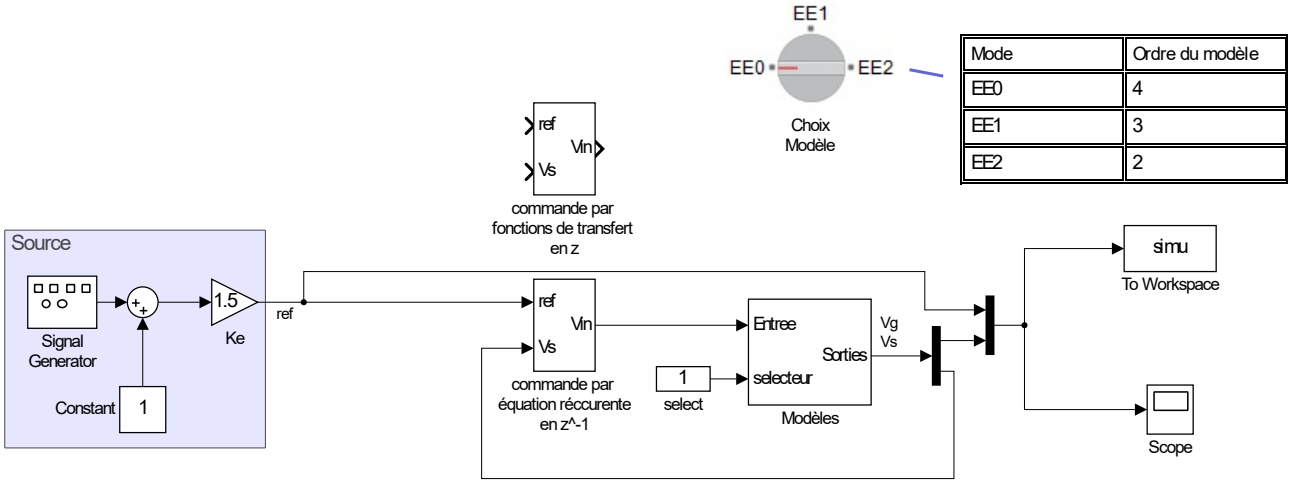


FIGURE 7.7 – *SIMULINK :* Modèle général

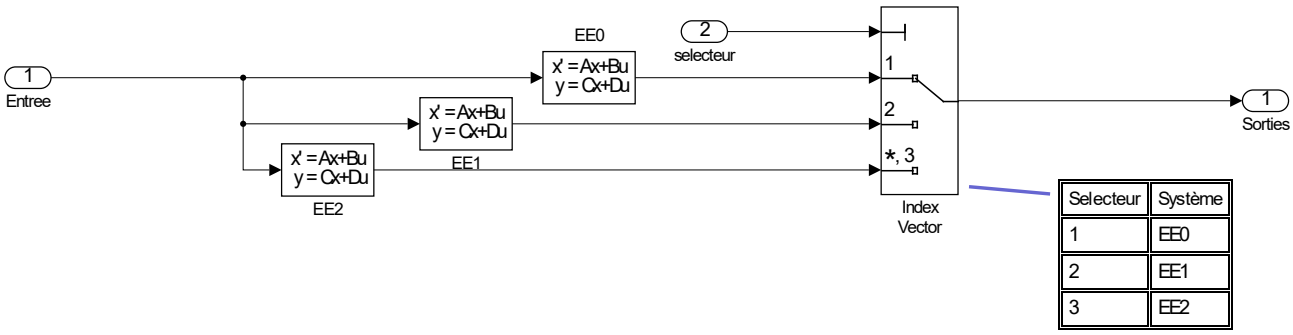


FIGURE 7.8 – *SIMULINK :* Subsystem des modèles

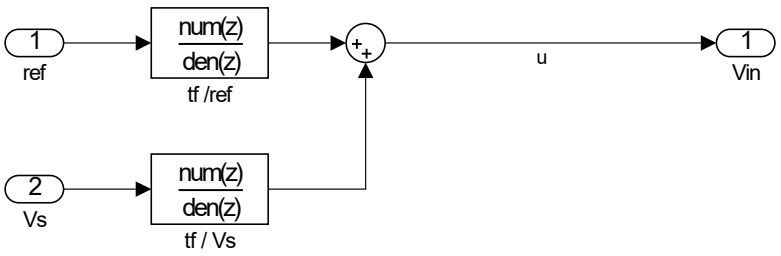


FIGURE 7.9 – *SIMULINK :* Subsystem de la commande par fonctions de transferts

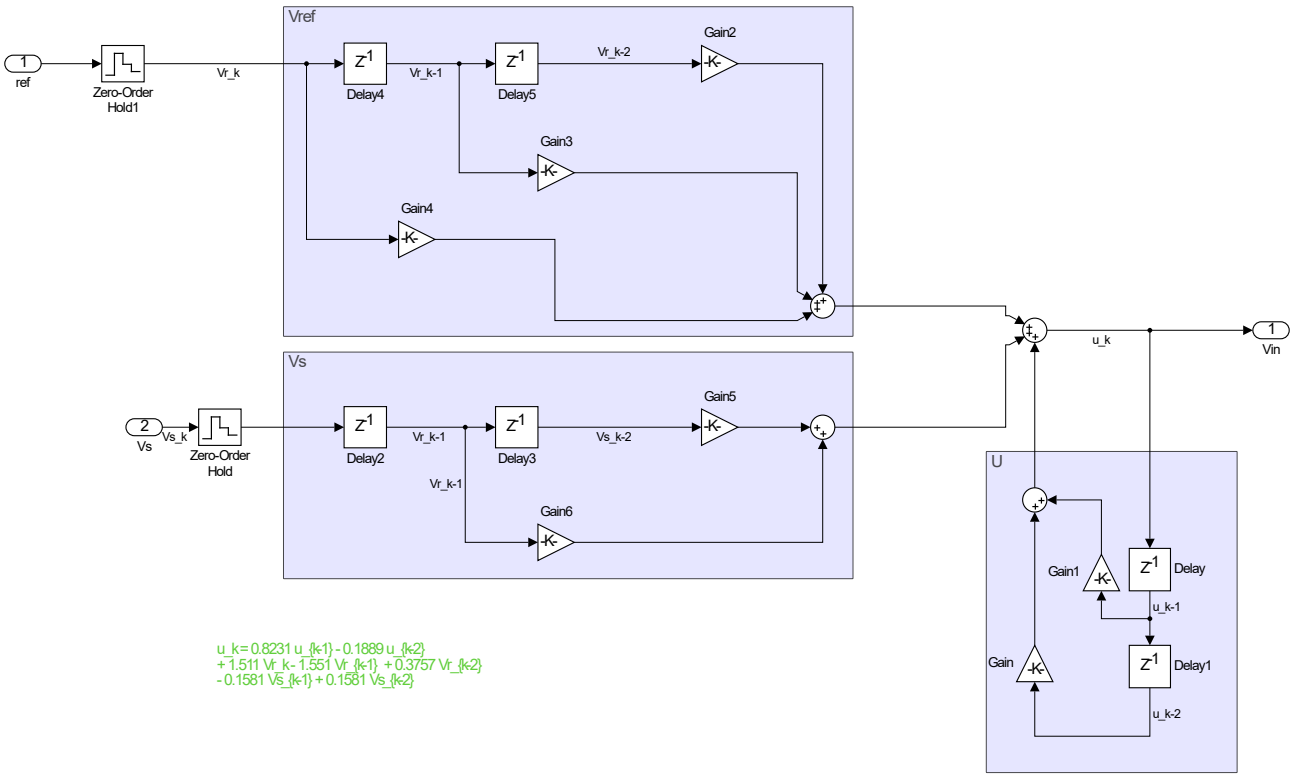


FIGURE 7.10 – *SIMULINK :* Subsystem de la commande par équation récurrente

Script Matlab

```
CommandeEE2
%%%%%%%%%%%%%%%%%%
%% simu modeles:
%%%%%%%%%%%%%%%%%%

%%EE0
% step
figure(1);
u=tf(1);
step(u,EE0.ee(1));
title('Reponse temporelle a un echelon de EE0')
legend('u(t)=1','Vs (t)','Location','southeast');
%% rampe
u = tf(1,[1 0]);
figure(2)
step(u,u*EE0.ee(1))
title('Reponse temporelle a une rampe de EE0')
legend('u(t)=t','Vs(t)','Location','southeast')
%% bode
figure(3)
bode(EE0.ee(1))
title('Bode de EE0')
legend('EE0')
%%
%%EE1
% step
figure(5);
u=tf(1);
step(u,EE1.ee(1),EE0.ee(1));
title('Reponse temporelle a un echelon de EE1')
```

```
31 legend('u(t)=1','EE1:Vs(t)','EE0:Vs(t)','Location','southeast')
32 %% rampe
33 u = tf(1,[1 0]);
34 figure(6)
35 step(u,u*EE1.ee(1),u*EE0.ee(1))
36 title('Reponse temporelle a une rampe de EE1')
37 legend('u(t)=t','EE1:Vs(t)','EE0:Vs(t)','Location','southeast')
38 %% bode
39 figure(7)
40 bode(EE1.ee(1),EE0.ee(1))
41 title('Bode de EE1')
42 legend('EE1','EE0')
43
44 %%EE2
45 % step
46 figure(8);
47 u=tf(1);
48 step(u,EE.ee(1),EE0.ee(1));
49 title('Reponse temporelle a un echelon de EE1')
50 legend('u(t)=1','EE1:Vs(t)','EE0:Vs(t)','Location','southeast')
51 %% rampe
52 u = tf(1,[1 0]);
53 figure(6)
54 step(u,u*EE1.ee(1),u*EE0.ee(1))
55 title('Reponse temporelle a une rampe de EE1')
56 legend('u(t)=t','EE1:Vs(t)','EE0:Vs(t)','Location','southeast')
57 %% bode
58 figure(7)
59 bode(EE1.ee(1),EE0.ee(1))
60 title('Bode de EE1')
61 legend('EE1','EE0')
62
63 %% Avec Commande_TD_EE2.slx, variable "simu"
64 model = 'EE0';
65 figure (8)
66     plot(simu.time,simu.signals.values(:,1),simu.time,simu.signals.values(:,2))
67     legend('ref(t)',sprintf('%s:Vg(t)',model),'Location','SouthEast');
68     title(sprintf('Asservissement de %s par une commande a temps
          discret',model));
69     xlabel('temps (s)');
70     ylabel('Tension (Volt)');
71 %% Affichage des valeurs de commandes a temps discret.
72 simuEE0=simu;
73 %%
74 simuEE1=simu;
75 %%
76 simuEE2=simu;
77 figure (9)
78     plot(simuEE0.time,simuEE0.signals.values(:,1),'k',...%ref
79          simuEE0.time,simuEE0.signals.values(:,2),'-.b',...%EE0
80          simuEE1.time,simuEE1.signals.values(:,2),'--r',...%EE1
81          simuEE2.time,simuEE2.signals.values(:,2),'g')%EE2
82
83     legend('ref(t)','EE0:Vg(t)','EE1:Vg(t)','EE2:Vg(t)','Location','SouthEast');
84     title(sprintf('Asservissement des modeles d''ordre 4, 3 et 2 \n par une
          commande a temps discret\n(Te=%f)',Te));
85     xlabel('temps (s)');
86     ylabel('Tension (Volt)');
```

Listing 7.1 – Script d'affichage

Annexe 5 - Code source C

Code source C

```
/*

 Generation d'un echelon de tension sur l'unite Numerique-Analogique
 Acquisition Analogique-Numerique du C167
 NR - 2016

 Pour compiler: gcc166 -Wall -g -m7 -o test-echelon test-echelon.c -ltpc167
 */

#include <c167.h>
#include <gnutrap.h>
#include <bool.h>
#include <commc167.h>
/* variables globales */
float V_out0;
unsigned int V_can0,V_cna0;

// consigne de la commande de 3 Volts
 int consigne=1;

// booleen pour savoir si la consigne est etat haut ou etat bas
int etat=0;

// variable contenant le temps ecoule relatif
int T=0;

// AJOUTS ETUDIANTS
/* Coefficients constants de la commande  */
/*
static float my0 =  1.421848;
static float my1 =  -2.529573;
static float my2 =  1.124884;
static float my4 = 0; // action Integrale

static float d1 =  1.000000;
static float d2 =  -1.667286;
static float d3 = 0; // action Integrale

static float mu0 =  -0.000045;
static float mu1 =  0.000000;
static float mu2 =  0.000045;
static float mu4 = 0; // action Integrale
*/
static float my0 =  1.510621;
static float my1 =  -2.140479;
static float my2 =  0.752529;
```

```
47
48 static float d1 =  -1.300908;
49 static float d2 =  0.434620;
50
51 static float mref0 =  0.000000;
52 static float mref1 =  -0.121555;
53 static float mref2 =  0.121555;
54 /* Variables d'etat */
55 float s0 ; // sortie k
56 float s1 ; // sortie k-1
57 float s2 ; // sortie k-2
58 //float s3 ; // sortie k-3      AJ integrateur
59
60 float ref0 ; // consigne k
61 float ref1 ; // consigne k-1
62 float ref2 ; // consigne k-2
63 //float ref3 ; //   consigne k-3    AJ integrateur
64
65 float mes0 ; // Y mesure k
66 float mes1 ; // Y mesure k-1
67 float mes2 ; // Y mesure k-2
68 //float mes3 ; // Y mesure k-3      AJ integrateur
69
70
71 // ******************** //
72
73 /* fonction d'initialisation de la conversion analogique-numerique */
74 void adc_io_init()
75 {
76     ADCON=0x0; /* conversion unique sur le canal */
77 }
78
79 /* fonction d'initialisation de la conversion numerique-analogique */
80 void dac_io_init()
81 {
82
83   DP6     = 0x001f;
84   ADDRSEL2  = 0x2008;
85   BUSCON2   = 0x8480;
86   ADDRSEL3  = 0x3008;
87   BUSCON3   = 0x8480;
88 }
89
90 /* fonction d'ecriture sur la voie 0 du DAC */
91 void dac0_output(unsigned w)
92 {
93   asm volatile("exts #0x0020,#1\n mov 0x0000,%0" : : "r" (w));
94 }
95
96 /* fonction d'ecriture sur la voie 1 du DAC */
97 void dac1_output(unsigned w)
98 {
99   asm volatile("exts #0x0030,#1\n mov 0x0000,%0" : : "r" (w));
100 }
101
102 /* fonction de generation d'un echelon sur la voie 0 du DAC */
103 /* sur interruption avec une periode T=2x3.36s */
104 TRAP(0x23,echelon_dac);
105 void echelon_dac() /* Lie a T3 */
106 {
107   if (etat==0)
```

```
108   {
109     /* Consigne etat HAUT*/
110     V_cna0 = 4095 ; /* valeur CNA 0 avec consigne pour etat haut */
111     V_out0 = V_cna0/4095*5 -5 ; /* tension recalculee */
112     etat=1;
113     SET_SFRBIT(P3.3);
114 
115     //printf_entier_console("CAN = %d ", V_can0);
116     //printf_entier_console("CNA = %d\n", V_cna0);
117     consigne = 3;
118 
119     //init_commande();
120   }
121   else
122   {
123     V_cna0 = 4095-2048 ; /* valeur CNA 0 a 0V pour etat bas */
124     V_out0 = 0; /* tension recalculee */
125     etat=0;
126     CLR_SFRBIT(P3.3);
127     //printf_entier_console("CAN = %d ", V_can0);
128     //printf_entier_console("CNA = %d\n", V_cna0);
129     consigne = 0;
130 
131     //init_commande();
132   }
133 
134   /* ecriture sur le DAC  ---Commentaire etudiant */
135   //dac0_output(V_cna0);
136 }
137 
138 /* ******* Init de la commande ******/
139 void init_commande()
140 {
141   // init consigne
142   s0 = 0;
143   s1 = 0;
144   s2 = 0;
145 //  s3 = 0; //    AJ integrateur
146   // init reference
147   ref0 = 0;
148   ref1 = 0;
149   ref2 = 0;
150 //  ref3 = 0;//   AJ integrateur
151   // init Y mesure
152   mes0 = 0;
153   mes1 = 0;
154   mes2 = 0;
155 //  mes3 = 0;//   AJ integrateur
156 }
157 
158 //float cTest=-5;  //  test can/cna
159 
160 
161 /* fonction de generation d'un echelon sur la voie 0 du DAC */
162 /* sur interruption avec une periode T=52ms */
163 TRAP(0x22,acquisition_adc);
164 void acquisition_adc() /* Lie a T2 */
165 {
166 
167   T=T+26; // periode de 52ms donc on ajoute 52ms au temps ecoule
168 
```

```
169   CLR_SFRBIT(ADCIR);           // reset du drapeau
170   SET_SFRBIT(ADST);            // activation sequence conversion
171
172   WAIT_UNTIL_BIT_SET(ADCIR);  // attente fin conversion Vg, 10us ecoulees
173   V_can0 = ADDAT & 0x03FF;      // lecture conversion Vg
174
175   // AJOUT ETUDIANTS
176
177   // commande
178 //  ref3 = ref2; //   AJ integrateur
179   ref2 = ref1;
180   ref1 = ref0;
181   ref0 = consigne;
182
183
184 //  mes3 = mes2; //   AJ integrateur
185   mes2 = mes1;
186   mes1 = mes0;
187   /* Adapatation mesure [0;5] vers [-5;5]*/
188   mes0 = V_can0 *(10.0/1023.0) -5.0;
189
190 //  s3 = s2; //   AJ integrateur
191   s2 = s1;
192   s1 = s0;
193
194   s0 =  -d1*s1 - d2*s2 /*- d3*s3*/ /* valeur des sorties precedentes */
195      + my0*mes0 + my1*mes1 + my2*mes2 /*+ my3*mes3*/ /* entree de y mesure
          */
196      + mu0*ref0 + mu1*ref1 + mu2*ref2 /*+ mu3*ref3;*/ /* entree de consigne
          */
197
198   // test CNA/CAN
199 /*  cTest+=.01;
200   if(cTest >5)
201     {
202       cTest=-5;
203       printf_entier_console("min%d\n",0);
204     }
205   s0 = cTest;
206 */
207
208   /* Adaptation de la sortie [-5;5] vers CNA [2048;4095] */
209   V_cna0 = (unsigned int)(2047.0/5.0*((s0 + 5.0)/2.0)+2048.0);
210   /* envoie du temps et de la valeur du CAN sur la ligne serie */
211   /* pour stockage et trace avec GnuPlot depuis le PC */
212   // Debug
213   printf_entier_console("T = %d ", T);
214   printf_entier_console("Consigne = %d ", consigne);
215   printf_entier_console("CAN = %u ", V_can0);
216   printf_entier_console("CNA = %u\n", V_cna0);
217   /*
218   // Test CAN/CNA
219   printf_entier_console("%d ", V_cna0);
220   printf_entier_console("%d\n", V_can0);
221
222   */
223   dac0_output(V_cna0);
224 }
225
226 /* initialisation Timer 3 pour generation echelon */
227 void timer_T3_init()
```

```
228 {
229   /* T3 en timer sans rechargement car periode pleine */
230   T3CON=0x407; // timer - resolution 51.2 us, comptage, periode 3.36s
231   T3=0; //valeur initiale du timer
232   T3IC=0xC; // IT Niveau 3 groupe 0
233   SET_SFRBIT(DP3.3);
234 
235 }
236 
237 /* initialisation Timer 3 pour generation echelon */
238 void timer_T2_init()
239 {
240   /* T2 en timer sans rechargement car periode pleine */
241   T2CON=0x01;// timer - resolution 1.6 us, comptage, periode 104 ms
242   T2=0; //valeur initiale du timer
243   T2IC=0x8; // IT Niveau 2 groupe 0
244 
245 }
246 
247 
248 void main()
249 {
250   init_ASC0_384();
251 
252   adc_io_init();
253   dac_io_init();
254   init_commande();
255   timer_T3_init();
256   timer_T2_init();
257 
258 
259   SET_SFRBIT(T3R); // activation timer
260   SET_SFRBIT(T3IE); // autorisation IT timer
261 
262   SET_SFRBIT(T2R); // activation timer
263   SET_SFRBIT(T2IE); // autorisation IT timer
264 
265   SET_SFRBIT(IEN); // autorisation IT generales
266 
267   do
268   {
269   }
270   while(1);
271 }
```

Listing 7.2 – Code source implémenter sur C167

Fichier coefficients commande

```
1    Parametres de la commande en temps discret
2 
3 static float my0 =  1.510621;
4 static float my1 =  -2.140479;
5 static float my2 =  0.752529;
6 
7 static float d1 =  -1.300908;
8 static float d2 =  0.434620;
9 
10 static float mref0 =  0.000000;
11 static float mref1 =  -0.121555;
```

```
12 static float mref2 =  0.121555;
```

Listing 7.3 – Code source implémenter sur C167