Projet I4

Intégration d'une loi d'ordonnancement dans un modèle

I4SE

Avril 2005

Charles
Sébastien
Christophe

Remerciements

Nous tenons à remercier tout particulièrement Monsieur R. Kocik, Responsable de la Majeure Systèmes Embarqués, qui nous a aidé tout au long de notre projet.

Nous remercions également Messieurs R. Hamouche, A. Réama et A. Fakri, enseignants et chercheurs du laboratoire COSI qui nous ont aidés pour la partie expérimentation de notre projet.

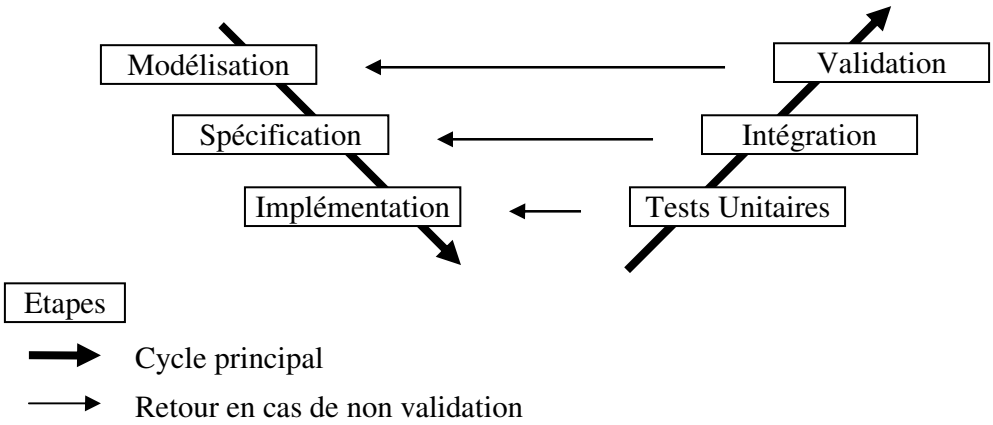
Merci aussi à A. Roche, ingénieur du laboratoire, qui nous a aidé à trouver et installer le matériel que nous avons utilisé.

SOMMAIRE

Remerciements 2
Introduction 4
I. Source des retards 5
A. Retards dus à l'ordonnanceur 5
B. Durée des tâches 7
II. La simulation sous Scicos 9
A. Le moteur à courant continu 9
B. Le système en boucle fermée 11
C. Le PID sous Scicos 12
D. Modélisation des tâches 13
E. Modélisation des retards sous Scicos 14
1. Gigue et Retard E/S fixes 14
2. Gigue et Retard aléatoires 15
3. Gigue et Retard exacts 16
III. Expérimentation 18
A. Schéma global du montage 18
B. Le Bloc moteur 19
C. L'Amplificateur 20
D. Le Générateur de signaux basse fréquence (GBF) 20
E. La Carte d'interface 20
F. La Carte PCI 1710 – Advantech 22
1. Port d'acquisition 22
2. Adaptations 22
G. Le programme RTAI 22
1. Pourquoi travailler avec RTAI ? 23
2. Programmation 23
H. Problèmes rencontrés 24
IV. Comparaison entre la simulation et la réalité 25
A. Étude des résultats avec le jeu 1 26
1. Cas Non préemptif 26
2. Cas Préemptif 32
B. Étude des résultats avec le jeu 2 36
V. Conclusion sur les résultats 40
VI. Programmes Additionnels 42
1. Calculus.c 42
2. Parse.c 42
3. Script.c 43
Conclusion 44
Bibliographie 45
ANNEXES 47
Bibliothèque *constantes.h* 48
Bibliothèque *constantes.h* 48
Fichier *task.c* 49
Fichier *calculus.c* 54
Shell linux *txt2dat* 57
Fichier *parse.c* 58
Script Matlab *script.m* 60
Expérimentation avec commande en Sinus 62

Introduction

Aujourd'hui, les systèmes embarqués de contrôles sont de plus en plus complexes et demandent des temps de développement de plus en plus longs. Ceci se répercute sur les prix des systèmes et, pour rester compétitifs, les entreprises doivent réduire leurs coûts.
Le cycle de développement classique d'un système est dit « en V » et se déroule selon le schéma suivant :



La modélisation consiste à formuler une loi de commande qui décrit mathématiquement un système de contrôle concret. Cette loi, dans la spécification, doit être découpée en tâches temps réel qui seront ensuite implémentées sur un calculateur.
Cependant, les automaticiens qui formulent les lois de commande négligent les retards induits par les temps de calculs et l'ordonnancement des tâches. Les performances du système sont ainsi dégradées et oblige les ingénieurs à corriger le modèle de la loi de commande pour permettre la validation du système de contrôle. Ceci entraîne des rebouclages du cycle de développement et donc une perte de temps et d'argent.

L'objectif de ce projet est de fournir des outils permettant d'évaluer au plus tôt les dégradations de l'implémentation sur la loi de commande. Il faut détecter ces retards pour les réintroduire dans le modèle de la loi de commande et ainsi fournir une simulation plus réaliste qui permet de valider le modèle au plus tôt.

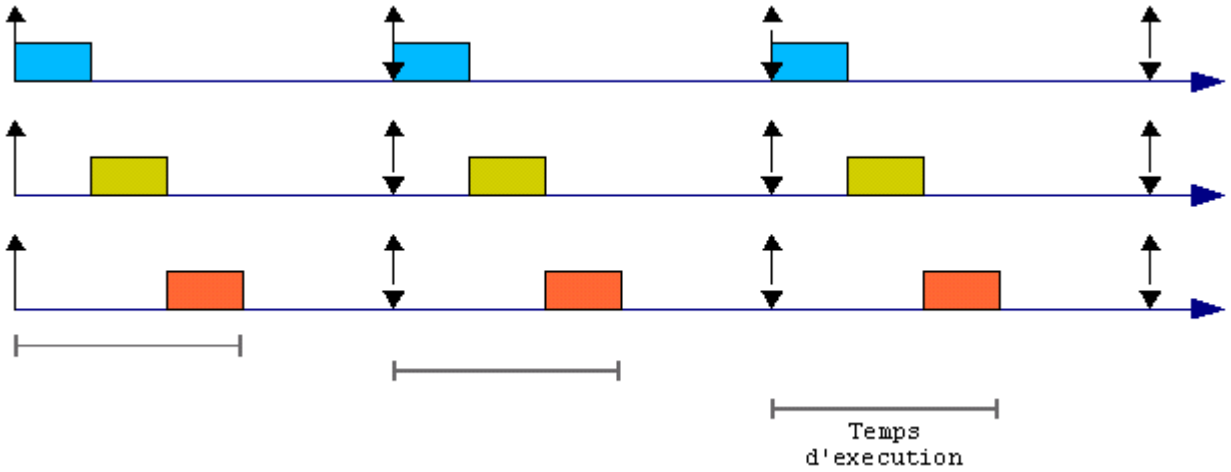
Dans un premier temps, nous développerons les éléments d'un ordonnancement qui peuvent provoquer des retards pour ensuite détailler comment il est possible de les introduire dans un modèle de simulation sous le logiciel Scicos. Ensuite, nous avons expérimenté l'influence des retards sur une loi de commande implémentée sur un Linux RTAI faisant un asservissement de vitesse sur un moteur à courant continu. Nous avons ainsi pu faire la comparaison entre les courbes obtenues théoriquement, avec et sans retards, et les courbes obtenues avec le système réel pour pouvoir tirer des conclusions sur la validité des modèles développés.

I. Source des retards

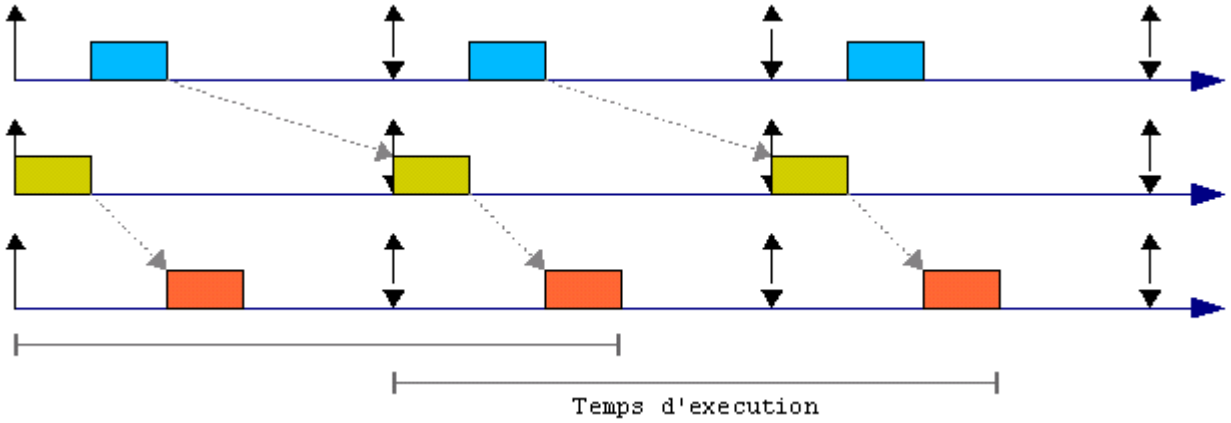
Dans cette partie nous allons expliquer les différentes sources de retards qui peuvent être crées à partir d'un ordonnancement de tâches temps réel. Pour illustrer ces causes des retards, nous allons prendre l'exemple de trois tâches temps réel qui dépendent l'une de l'autre. L'ordre des tâches est : bleu, jaune, rouge et leur durée n'est pas forcement fixe.

A. Retards dus à l'ordonnanceur

Si tout se déroule parfaitement, les trois tâches seront exécutées l'une après l'autre, sans aucun retard. Le temps d'exécution sera constant et le débit de données constant:



Mais si l'ordre des tâches devient différent, le temps d'exécution varie:



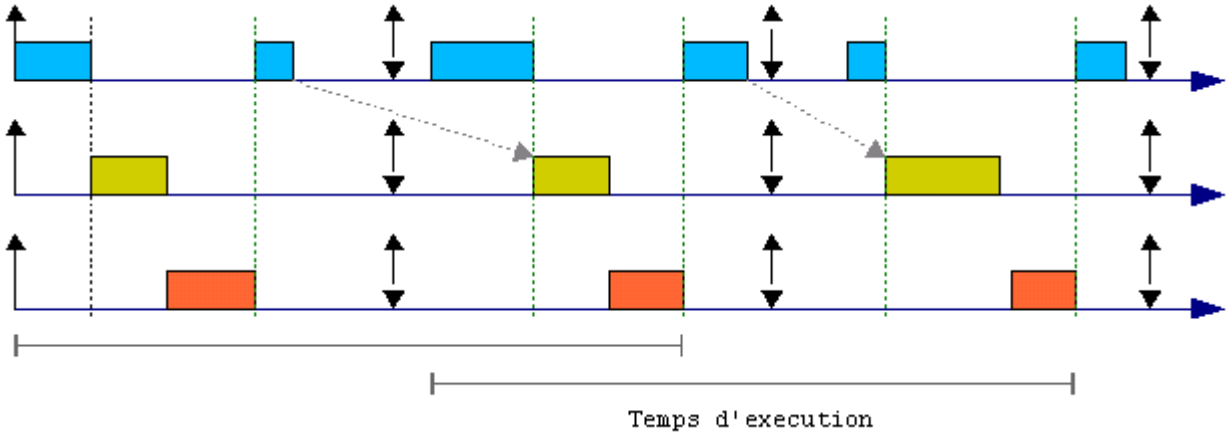
On remarque qu'en modifiant juste l'ordre des tâches, on augmente beaucoup le temps d'entrée sortie. Néanmoins nous obtenons toujours une sortie toutes les secondes mais cette sortie représente l'entrée de la période précédente.

L'ordonnanceur peut également introduire, pour chaque tâche, une gigue due au temps de calcul pour ordonnancer toutes les tâches:

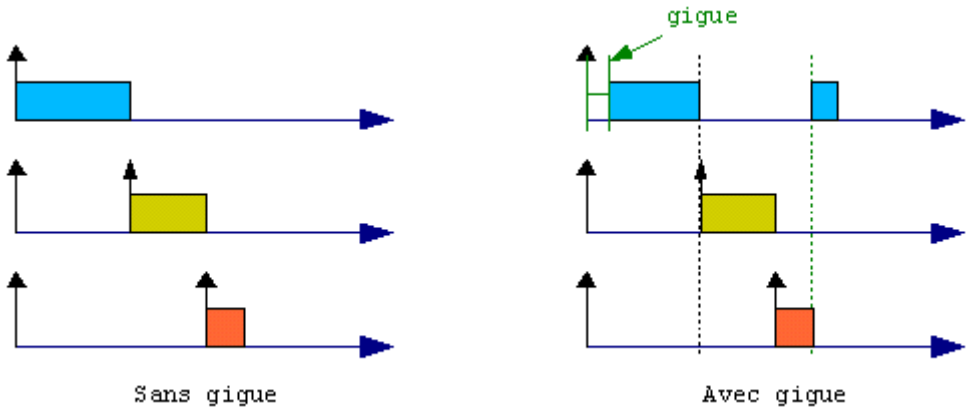
Gigue de démarrage

Cette gigue aura pour conséquence de faire varier le débit de données. Généralement cette gigue est négligeable devant les périodes des tâches. Mais si le nombre de tâches est important, ces retards vont s'additionner et créeront un retard non négligeable.

Il est possible également que l'ordonnanceur préempte les tâches. Le temps d'exécution peut alors augmenter énormément:



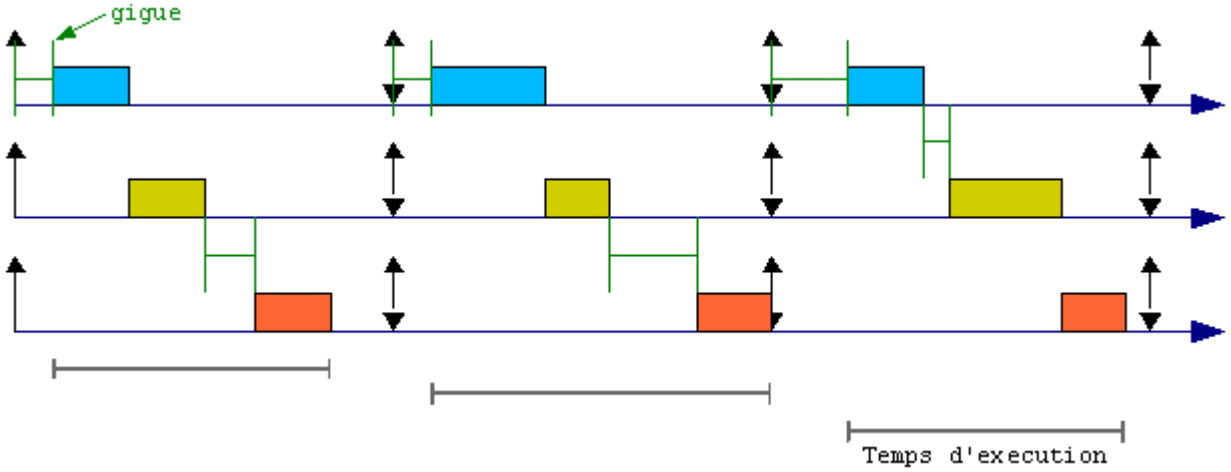
Il est possible qu'un jeu de tâches soit ordonnancable en théorie mais qu'en pratique il ne le soit pas. Si la gigue ajoute un léger retard, la date de terminaison d'une tâche peut être beaucoup retardée et tout l'ordonnancement est modifié. Par exemple en supposant que les tâches jaunes et rouges soient plus prioritaires:



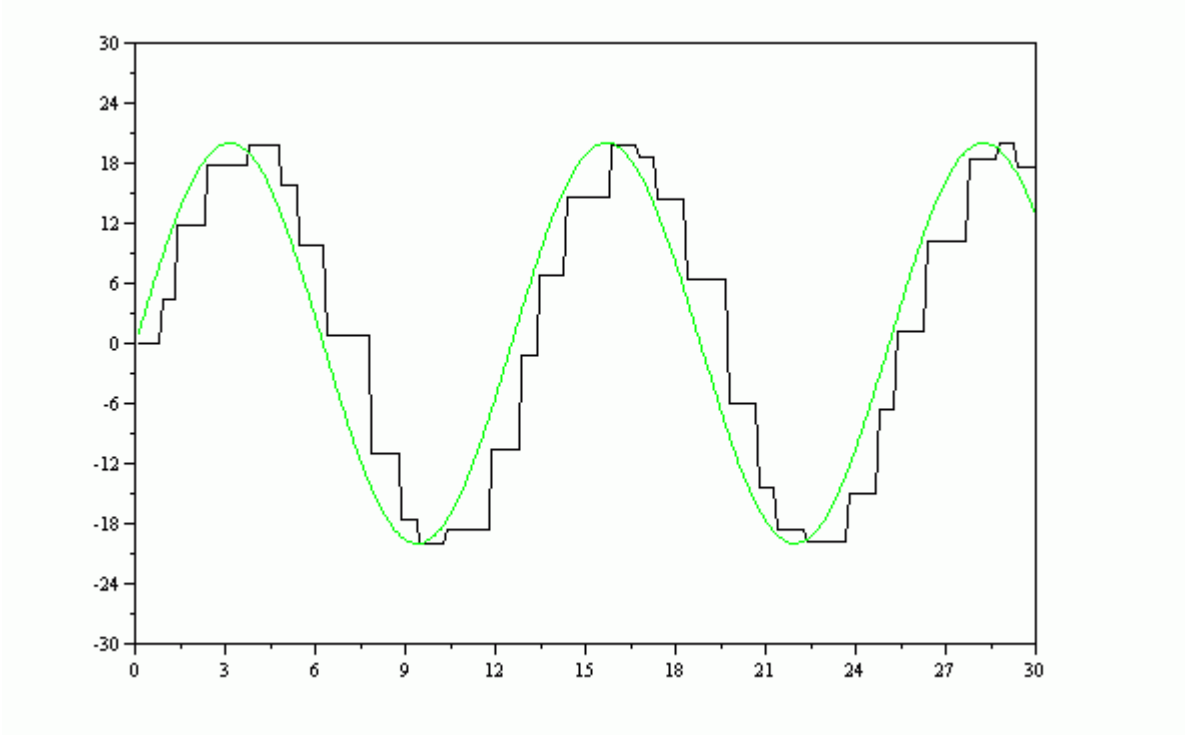
La tâche bleue est préemptée et peut entraîner un décalage de tout le reste de l'ordonnancement.

B. Durée des tâches

Les tâches peuvent effectuer des calculs ou des opérations d'entrée sortie qui n'ont pas une durée fixe dans le temps. Ces variations ainsi que la gigue de démarrage peuvent faire varier le temps d'entrée sortie ainsi que le débit de données:

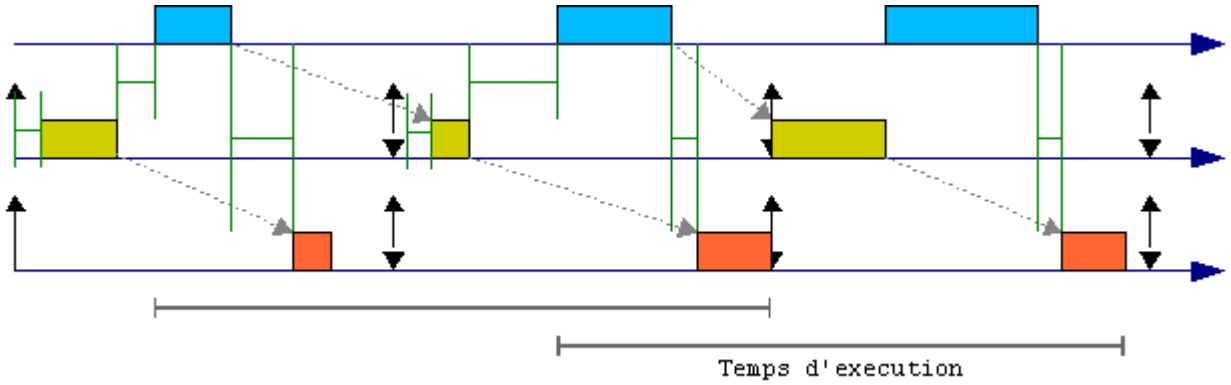


Pour illustrer l'effet de ces retards, nous allons attribuer des rôles à ces tâches. La tâche bleue fait une acquisition, la tâche jaune fait un calcul et la tâche rouge fait une sortie. En supposant que l'on souhaite traiter un signal sinusoïdal avec un simple gain de 1, on pourrait obtenir le résultat suivant:

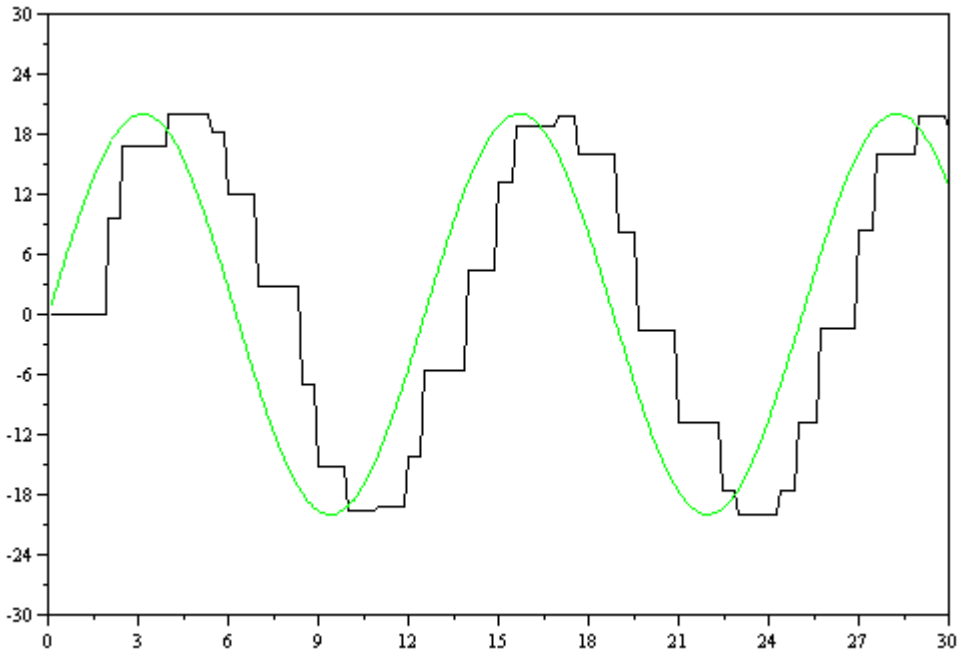


Le signal est déformé car les échantillons ne sont pas exactement périodiques. Néanmoins la courbe conserve son aspect général.

Si en plus l'ordre des tâches varie, un retard d'entrée sortie s'ajoute aux retards précédents:



Dans ce cas l'exemple du sinus devient :



Le signal est toujours déformé à cause de la gigue, et de plus il est décalé à cause du retard d'entrée sortie dû à l'ordonnancement.

II. La simulation sous Scicos

Scicos est une extension du logiciel de calcul numérique SciLab (http://scilabsoft.inria.fr) qui permet de modéliser des systèmes dynamiques sous forme de schémas blocs. A partir d'une loi de commande et d'un modèle mathématique, il est possible de prévoir l'évolution des caractéristiques d'un système. Par la suite, cette loi de commande peut être implémentée dans un calculateur sous forme de différentes tâches temps réel et interfacée pour obtenir les résultats souhaités. Cependant, une loi basique ne prend pas en compte les retards qui peuvent être dus à l'ordonnancement ou aux temps de calcul et ceux-ci peuvent entraîner des erreurs importantes dans le comportement du système global ; pouvant aller jusqu'à une instabilité de ce dernier.
L'intérêt de Scicos pour notre projet est qu'il possède une gestion efficace des événements. Chaque bloc d'un schéma peut être déclanché à un temps précis via l'envoi d'un événement. Ainsi, il devient facile de simuler un ordonnancement en déclanchant les tâches temps réel aux moments désirés.

Dans notre projet, nous avons étudié l'influence des retards sur un asservissement de vitesse d'un moteur à courant continu. La vitesse est contrôlée grâce à un correcteur Proportionnel Intégral Dérivé (PID). Nous avons donc réalisé sous Scicos le modèle d'un PID sur un moteur pour pouvoir comparer les réponses du système.

Il faut donc, pour pouvoir réaliser une modélisation correcte, connaître les caractéristiques du moteur sur lequel nous effectuons le contrôle.

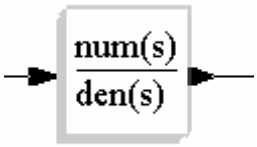
A. Le moteur à courant continu

Un moteur est un système dynamique assimilable à la somme de trois éléments : une résistance, une inductance et une source de tension issue de la force électromotrice.
La vitesse en régime continu est proportionnelle à la tension aux bornes du moteur, mais en régime transitoire, le moteur réagit comme un système du premier ordre dont l'équation est :

$$\frac{Gm}{1+Tm*p} \text{ en Laplace}$$

Avec Gm le gain du moteur en vitesse en rad.s^{-1} /V
Tm la constante de temps mécanique

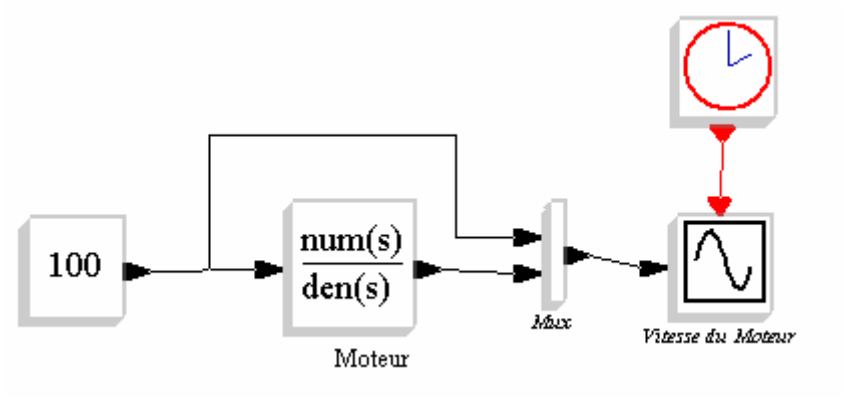
Cependant ce modèle ne prend pas en compte les perturbations dues à l'inductance du moteur ainsi que les forces de frottements qui sont négligeables, à vide, par rapport à la force mécanique.
Par expérimentation, nous avons trouvé un Tm de 2.6s qui correspond au temps que met le moteur pour atteindre 63% de la vitesse demandée.
Sous Scicos, le moteur est simplement représenté par un bloc « Continous Tranfert Function »



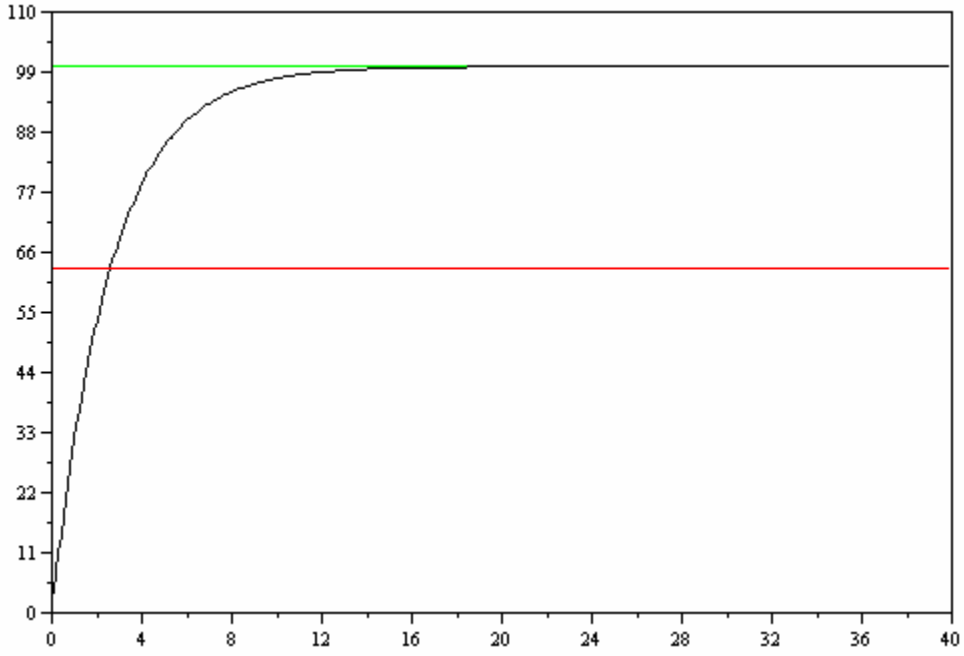
Dans notre simulation, le gain du moteur en vitesse n'est pas utile car il est possible de donner une consigne directement en rad.s^{-1}. La fonction de transfert du moteur devient alors :

$$\frac{1}{1+Tm*p}$$

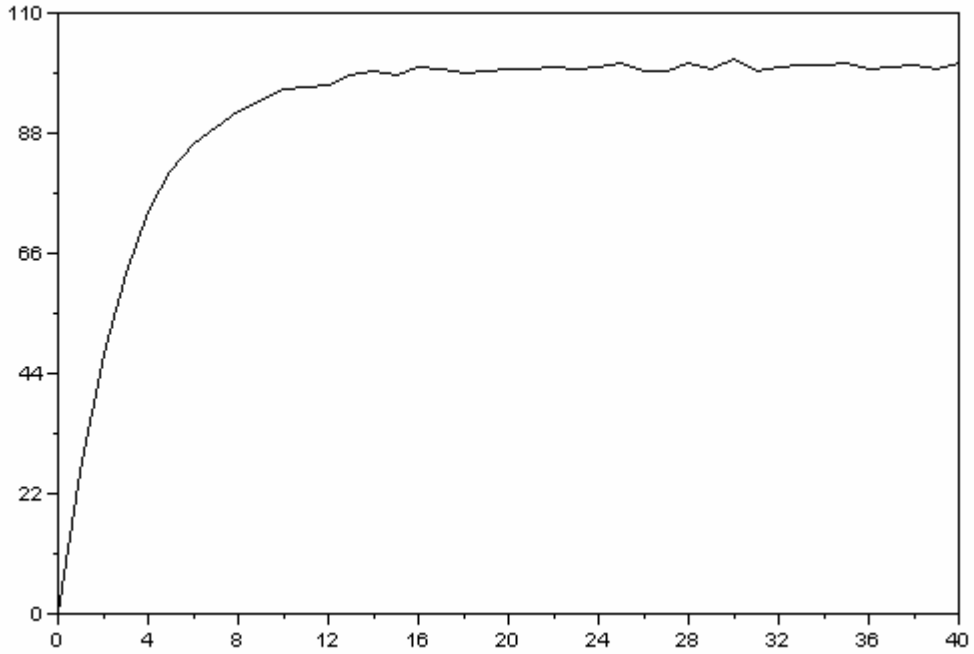
Voici la modélisation du moteur en boucle ouverte :



En boucle ouverte, on obtient en simulation, pour une commande de 100 rad.s^{-1}, la réponse suivante :



Grâce à la carte d'acquisition, dont nous détaillerons le fonctionnement plus loin, nous obtenons la réponse suivante. Cependant, le calcul du Tm a été effectué sur un oscilloscope pour éviter les problèmes de retards qui sont le cœur de ce projet. La représentation suivante ne sert donc que d'illustration :

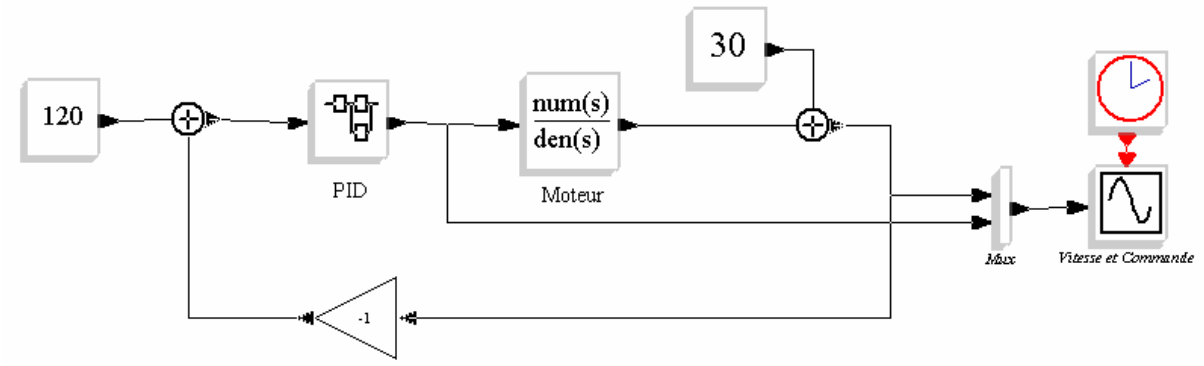


On vérifie ainsi que la modélisation du moteur en simulation semble juste.

Pour pouvoir évaluer l'influence des retards sur un ordonnancement, la vitesse du moteur est contrôlée par un PID qui est également modélisé sous Scicos.

B. Le système en boucle fermée

On réalise donc un système en boucle fermée sous Scicos. Dans l'exemple ci-dessous, l'objectif de la vitesse est de 120 rad.s^{-1} et pour que l'étude se fasse sur la partie de fonctionnement linéaire du moteur, on rajoute une vitesse initiale de 30 rad.s^{-1}.



Modèle global du système

C. Le PID sous Scicos

Un PID est une méthode de contrôle qui permet de corriger un paramètre d'un système bouclé. Dans notre expérimentation, le contrôle s'effectue sur la vitesse d'un moteur à courant continu. En boucle fermée, on calcule l'erreur entre la consigne et la vitesse, puis, à travers le PID, on envoie une nouvelle commande.

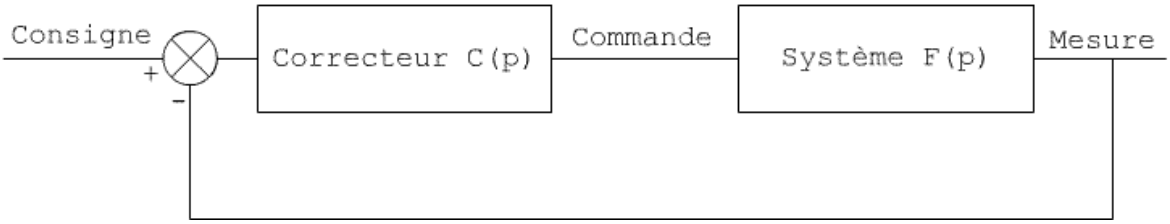


Schéma théorique du PID

En Laplace, l'équation qui définit la commande est :

$$C(p) = \frac{u(p)}{\varepsilon(p)} = K + \frac{Ki}{p} + Kd \times p$$

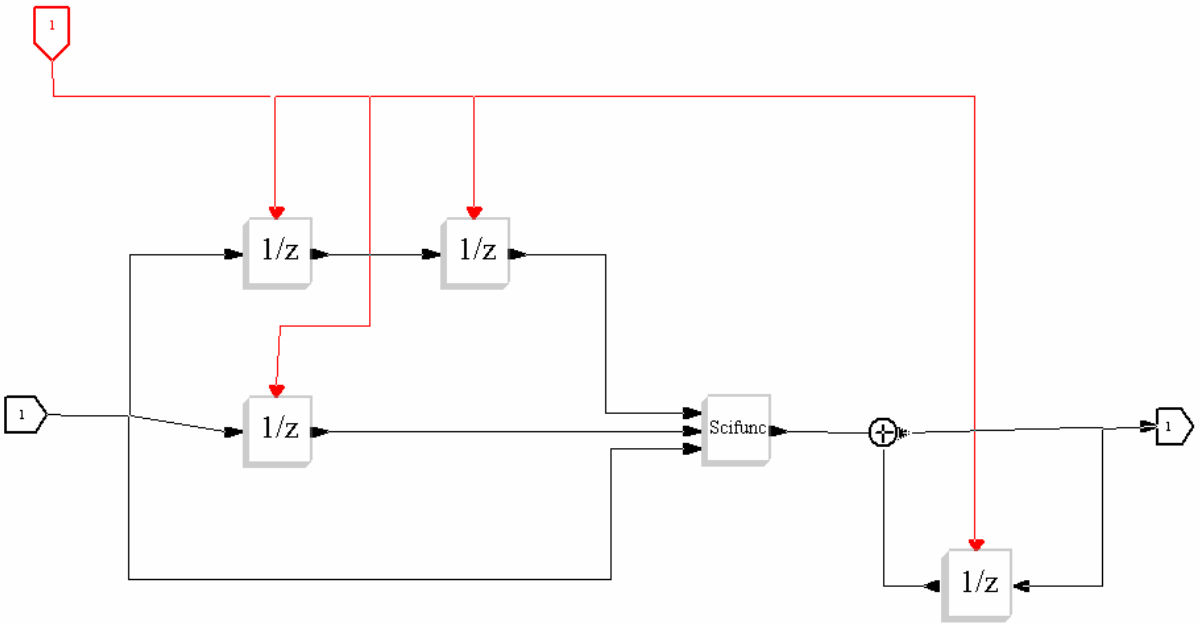
K, Ki et Kd sont des coefficients qui permettent de pondérer des différentes parties du PID.
Cependant, comme le calcul doit être effectué en discret pour qu'il soit implémenté sur un calculateur, on pose

$$p = \frac{1 - z^{-1}}{Te}$$ avec Te la période d'échantillonnage

On obtient ainsi l'équation discrète de la commande :

$$u(k) = u(k-1) + K(\varepsilon(k) - \varepsilon(k-1)) + Ki \times \varepsilon(k) \times Te + Kd \times \frac{\varepsilon(k) - 2 \times \varepsilon(k-1) + \varepsilon(k-2)}{Te}$$

Pour notre projet, nous avons choisit K=0.9, Ki=0.9 et Kd=0. La partie dérivée est ainsi ignorée car elle a pour défaut d'amplifier les bruits de mesures et que, pour un asservissement de vitesse, elle n'est pas utile.
Sous Scicos, le calcul du PID peut s'effectuer comme suit :

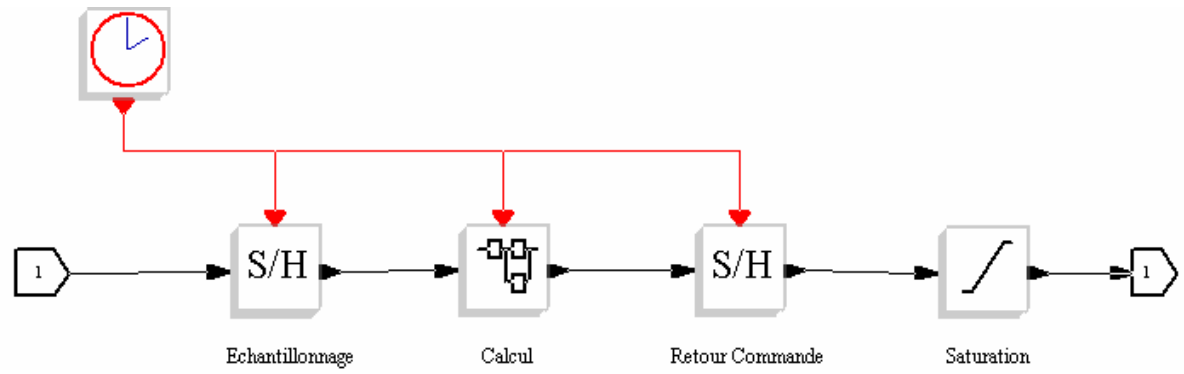


Modèle de calcul du PID

Code du bloc Scifunc :
y1=K(u3-u2)+Ki*u3*Te+Kd*(u3-2*u2+u1)/Te;*

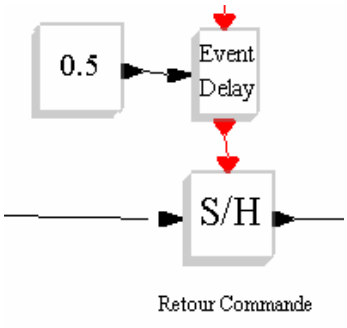
D. Modélisation des tâches

Pour pouvoir effectuer une analyse des retards, le programme RTAI est composé de 3 tâches temps réels de durées différentes : 1 tâche qui calcule le PID et 2 tâches qui viennent perturber le fonctionnement de la première, soit en la préemptant, soit en retardant sa date de démarrage. Ceci permet d'avoir des variations sur la gigue et sur le temps entre l'entrée et la sortie du PID. Chacune des tâches possède une période d'exécution et une durée fixe.
Sous Scicos, la tâche du PID est représentée grâce à 3 blocs principaux, l'échantillonnage, le calcul du PID et le retour commande.
Voici le schéma :



Modèle sans retards

L'horloge est un générateur d'événement périodique. Ainsi, il est possible de simuler un échantillonnage toutes les secondes par exemple en mettant 1 dans les propriétés de l'horloge.
Dans l'exemple ci-dessus, le temps de calcul n'est pas pris en compte dans la simulation. Pour qu'il soit considéré, il suffit de rajouter un bloc « event delay » au dessus du bloc de commande.



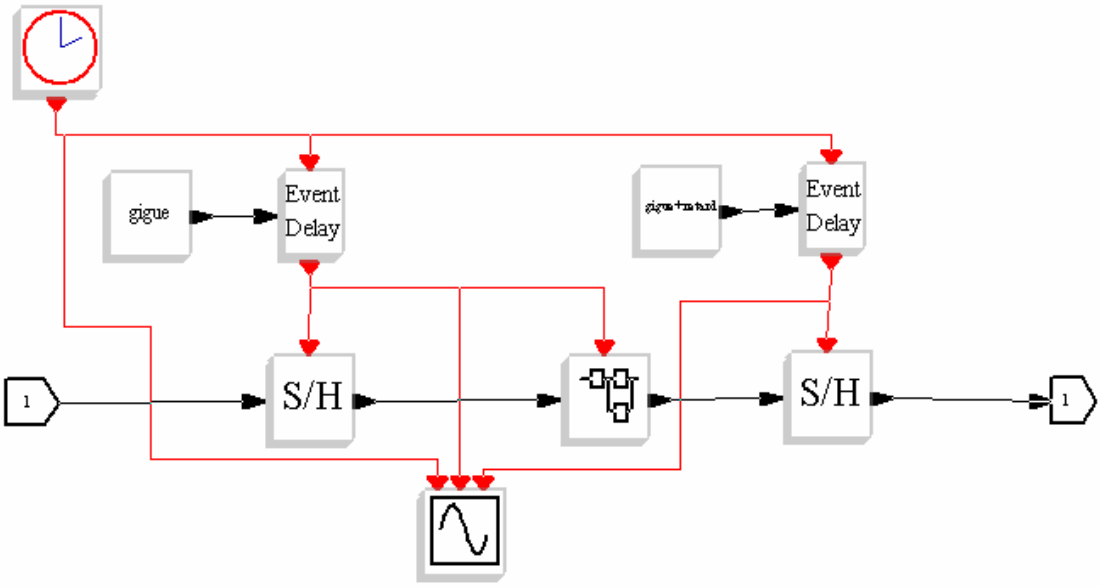
En mettant un retard d'événement de 0.5 seconde, on simule une tâche de durée équivalente. On peut ainsi grâce à ces bases modéliser les différents modèles de retards vus précédemment.

E. Modélisation des retards sous Scicos

A partir d'un ordonnancement, on peut extraire différentes données dont la gigue et le retard E/S des tâches. Il y a plusieurs moyens de les introduire dans un modèle.

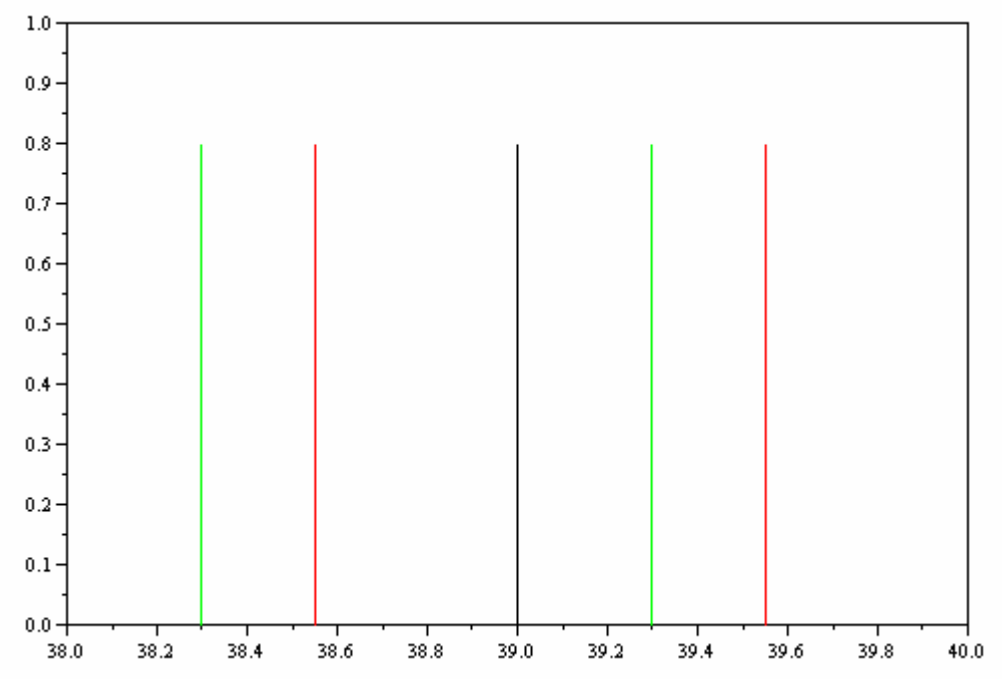
1. Gigue et Retard E/S fixes

On peut réintroduire dans le modèle des retards fixes. Les valeurs de ces retards peuvent correspondre à la valeur moyenne ou à la valeur maximum calculées avec l'ordonnancement.



Modèle avec retards fixes

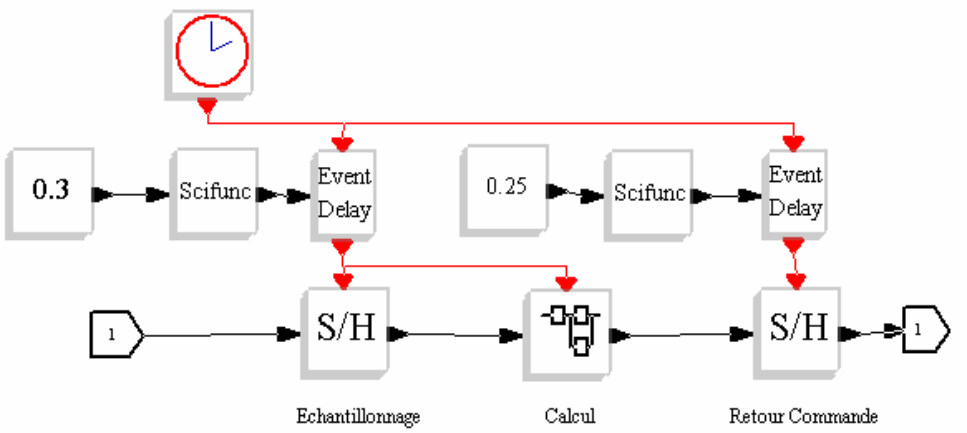
En mettant un observateur d'événement, on obtient un ordonnancement avec des temps de début et de fin de tâches constants ainsi que les périodes d'échantillonnage.



Visualisation de l'exécution du PID en simulation

2. Gigue et Retard aléatoires

Avec un ordonnancement classique, les retards sont rarement constants. Ainsi, il est possible de rajouter une partie aléatoire sur les temps. Ceci peut se faire en utilisant un bloc « Scifunc » qui permet d'utiliser du code Scilab sur un signal.



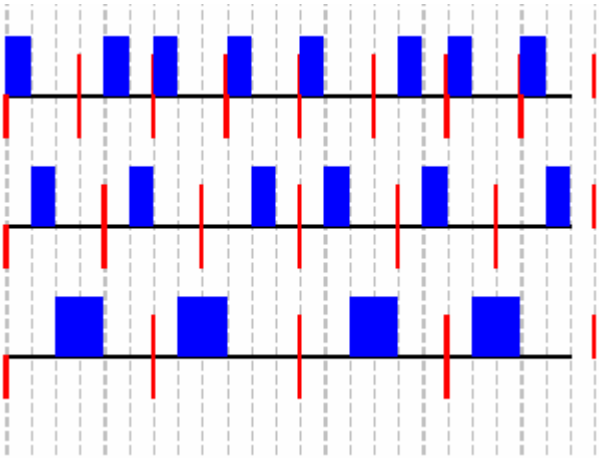
Modèle avec retards aléatoires

Le code du bloc Scicos utilise la fonction rand() qui permet d'obtenir un nombre aléatoire avec une distribution uniforme ou normal.

Code du bloc Scifunc :
*y1=u1+rand(1,'uniform')*u1;*

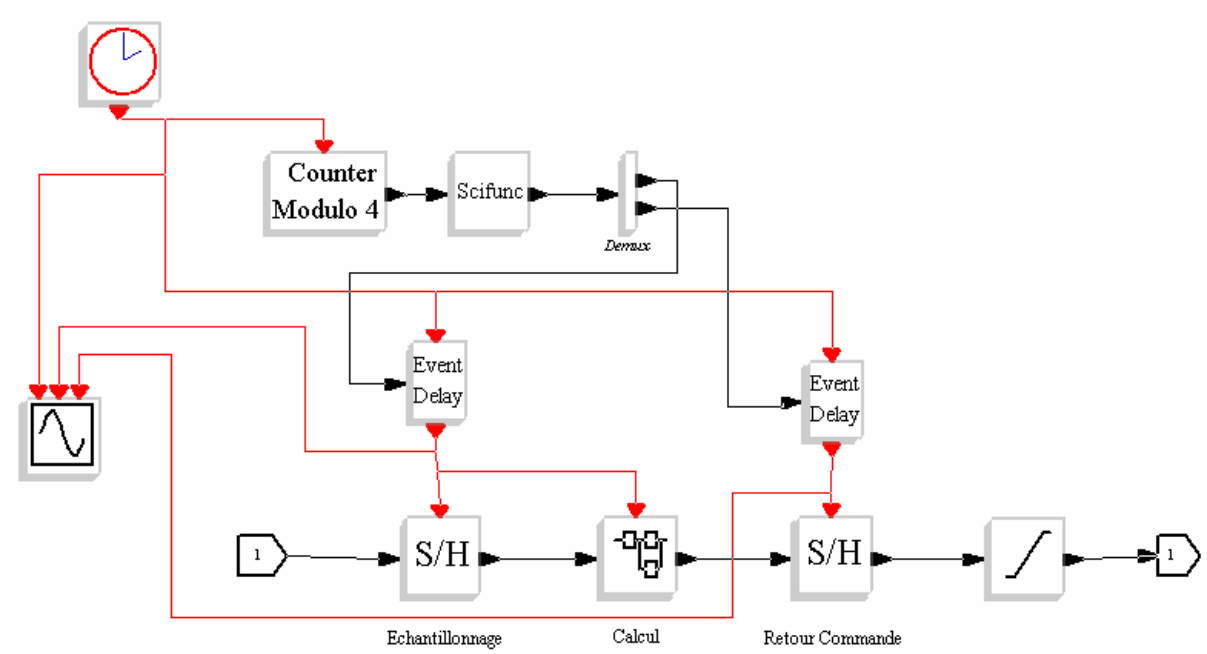
3. Gigue et Retard exacts

Pour créer un modèle qui soit le plus proche de la réalité, la solution est de spécifier, pour chaque période, les délais de gigue et d'E/S dus à l'ordonnancement.
Pour cela, il faut d'abord identifier la période absolue des tâches, c'est-à-dire le temps à partir duquel l'ordonnancement se répète à l'identique par rapport au début.
Exemple d'ordonnancement :



Ordonnancement de trois tâches

Dans cet ordonnancement, la période absolue est de 12 car après la 12ème unité de temps, les tâches reprennent le même ordre d'exécution qu'au début et ainsi les retards sont les mêmes. Sous Scicos, cette période est utilisée dans un compteur. Ensuite, un bloc « scifunc » identifie la période courante et attribut les temps de retards sur les « event delay ».

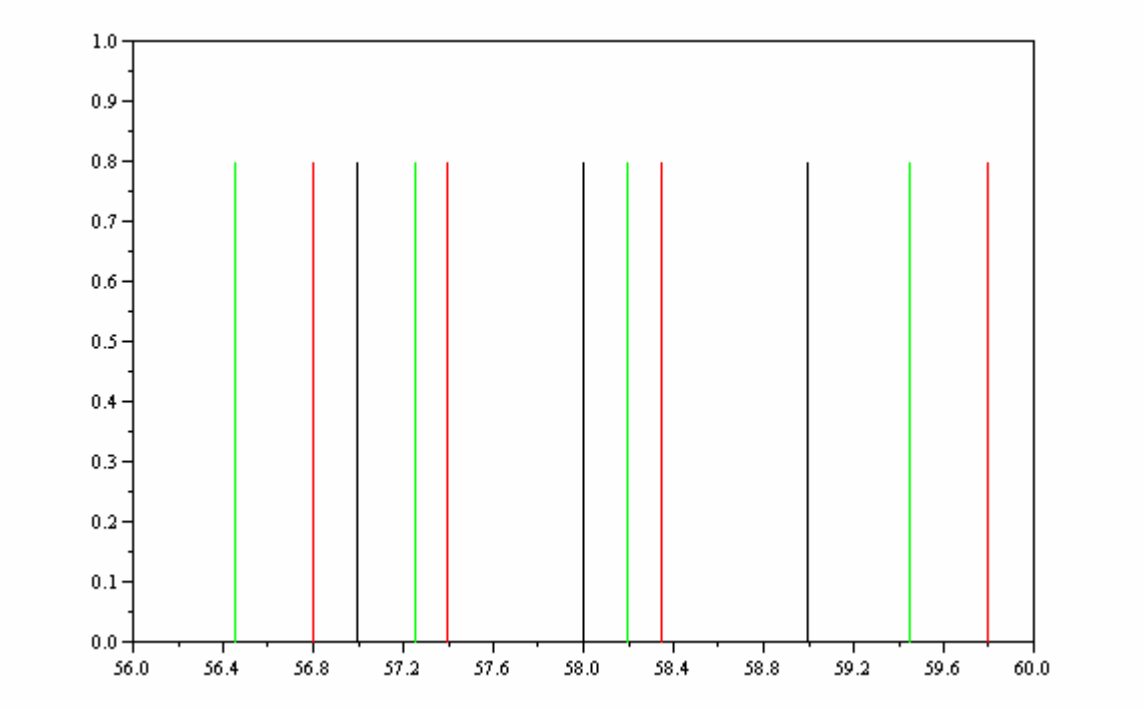


Modèle avec retards exacts

Code du bloc Scifunc :

```
select u1
  case 0 then y1(1) = 0.45 ; y1(2) = y1(1) + 0.15 + 0.2;
  case 1 then y1(1) = 0.25 ; y1(2) = y1(1) + 0.15 ;
  case 2 then y1(1) = 0.2   ; y1(2) = y1(1) + 0.15 ;
  case 3 then y1(1) = 0.45 ; y1(2) = y1(1) + 0.15 + 0.2;
  else y1(1) = 0; y1(2)=0;
end;
```

Voici un exemple des temps d'exécution de la tâche PID correspondant au code précédent :



Période, Début, Fin

Visualisation de l'exécution du PID en simulation

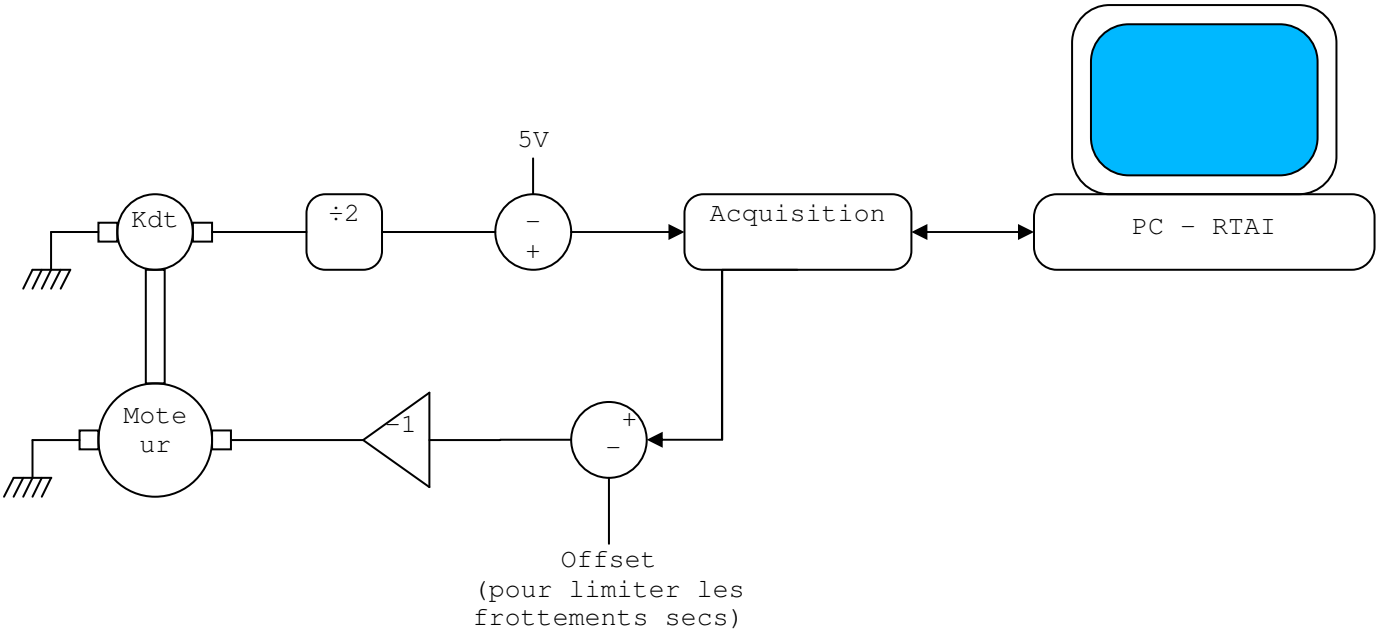
On peut voir que le débit n'est pas constant car le temps entre deux sorties varie.

III. Expérimentation

Pour pouvoir faire évaluer de façon concrète l'influence des retards sur une loi de commande, nous avons implémenté un asservissement de vitesse sur un moteur à courant continu sous Linux RTAI par l'intermédiaire d'une carte d'interface qui effectue l'acquisition de la vitesse et qui fait le retour de la commande calculé par notre programme. Le contrôle tachymétrique est effectué par une tâche temps réel qui effectue une correction Proportionnel Intégrale sur la vitesse. Pour que les temps de retards puissent varier, deux autres tâches ont été créées pour perturber le déroulement du calcul.

A. Schéma global du montage

Voici le schéma simplifié du montage :



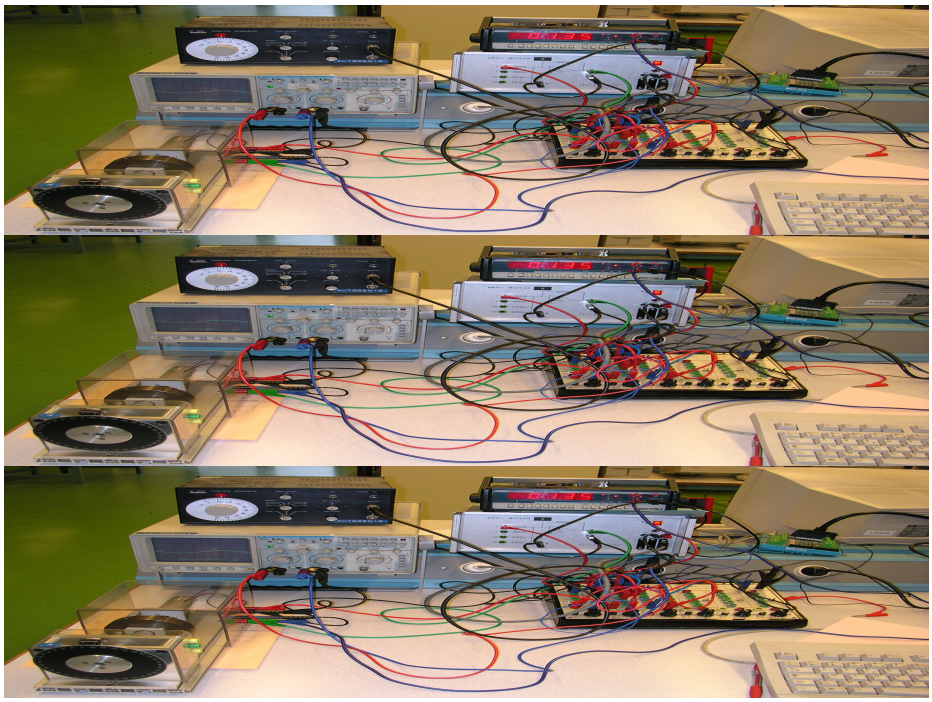
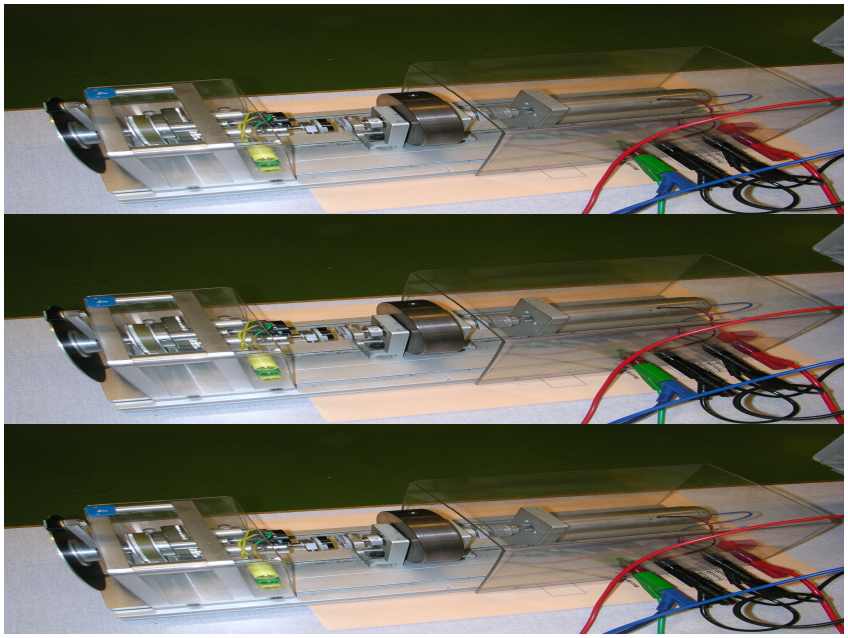


Photo du montage

B. Le Bloc moteur

Le bloc moteur est constitué d'un moteur à courant continu et d'un capteur tachymétrique qui nous permet de connaître la vitesse du moteur en temps réel. Nous obtenons la vitesse du moteur en multipliant la tension obtenue en sortie du capteur par une constante nommée *Kd* qui vaut *Kd-0.0497*.

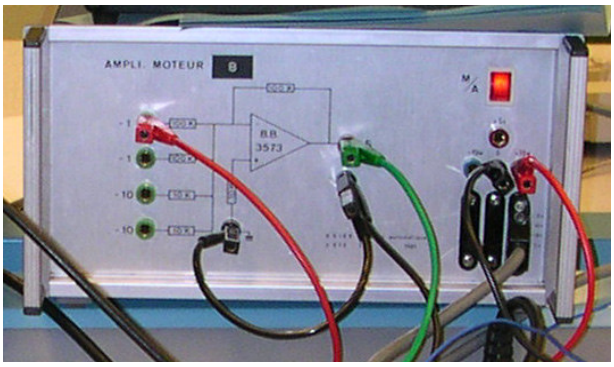


Moteur à courant continu

Afin de mieux contrôler le moteur, nous avons effectué des essais en boucle ouverte, ce qui nous a permis de mesurer les caractéristiques du moteur. Nous en avons ainsi déduit son gain *Gm* qui vaut *Gm*=26,15 et son temps de réponse *Tm* =2,6.

C. L'Amplificateur

Nous disposons d'un amplificateur à gain négatif, qui donne de la puissance à notre signal de commande, avant d'être envoyé au moteur.



Amplificateur

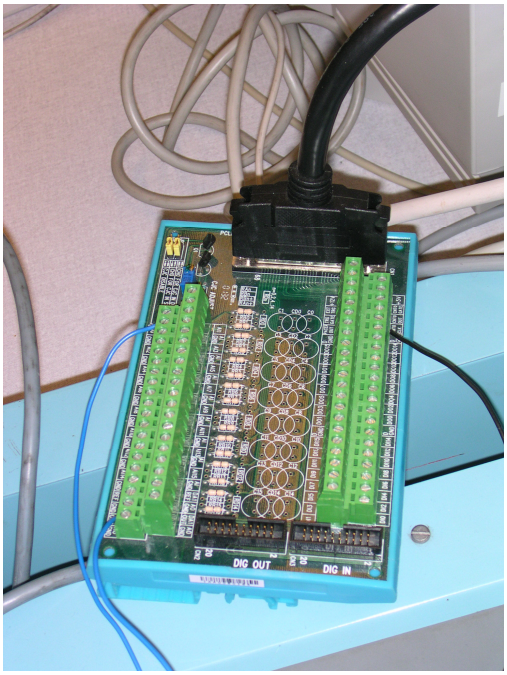
D. Le Générateur de signaux basse fréquence (GBF)

Pour vérifier le bon fonctionnement de notre asservissement de vitesse, nous utilisons un GBF pour générer un signal sinusoïdal que nous appliquons pour perturber la commande du moteur.

Générateur basse fréquence (GBF)

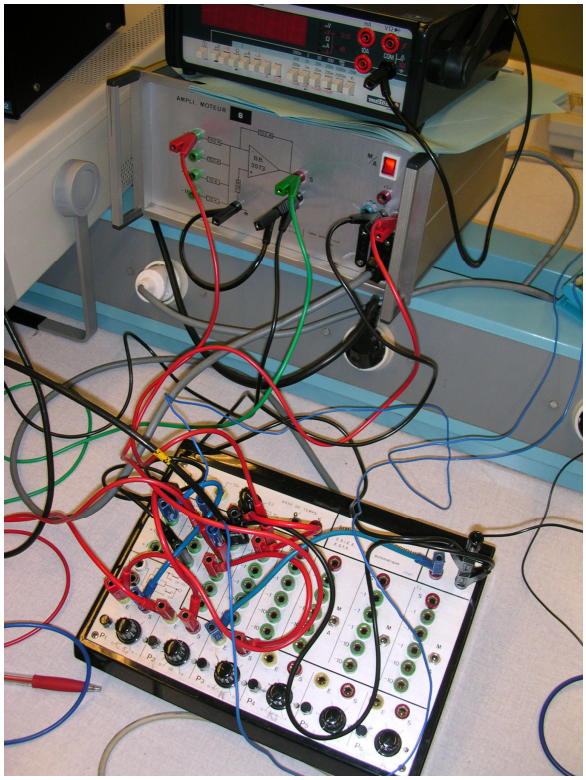
E. La Carte d'interface

Afin de faire l'interface entre la maquette et le PC, nous disposons d'une carte d'acquisition Advantech 1710 insérée dans le PC sur un port PCI, reliée à notre montage par une carte d'interface.



Carte d'interface

De plus, nous avons besoin d'adapter nos signaux avant de les envoyer à l'ordinateur. Ces adaptations sont réalisées à l'aide d'un pupitre de commande analogique qui dispose de plusieurs sommateurs et de plusieurs potentiomètres, pour nous permettre de générer des tensions diverses.


Pupitre de commande analogique

F. La Carte PCI 1710 – Advantech

1. Port d'acquisition

Nous n'utilisons que deux ports d'entrée/sortie de la carte, un port pour l'acquisition de la vitesse du moteur et un second pour l'envoi de la commande.

Pour l'acquisition de la vitesse du moteur nous utilisons le port AI1 qui est relié à un convertisseur analogique-numérique (CAN).

La commande est fournie au moteur par le port DAO_0 de la carte.

2. Adaptations

La carte d'acquisition ne supporte en entrée que des tensions comprise entre -5V et +5V, or nous récupérons, en sortie du capteur tachymétrique, une tension allant de 0 à 12V. Pour palier à ce problème, nous effectuons une division de tension pour obtenir une plage de tension comprise entre 0 et 6V. Nous retranchons ensuite 5V à la tension récupérée, pour enfin l'envoyer à l'ordinateur.
Une fois la tension récupérée au niveau du programme de calcul, nous multiplions la tension par deux, pour obtenir la tension réelle à la sortie du capteur.
Nous obtenons ainsi des valeurs numériques comprise entre 0 et 4095, 0 correspondant à -5V en entrée, soit 0V en sortie du capteur, et 4095 correspondant à +5V en entrée de la carte, soit 12V en sortie du capteur.

En ce qui concerne la commande, la carte d'acquisition est capable de générer des tensions comprises entre 0 et 10V, de ce fait, nous n'avons pas besoin d'adapter la tension pour la commande. A niveau du programme informatique, la valeur numérique à envoyer au convertisseur numérique-analogique (CNA) est comprise elle aussi entre 0 et 4095, 0 correspondant à 0V en sortie et 4095 correspondant à 10V en sortie de la carte.

G. Le programme RTAI

Nous utilisons dans le PC un système d'exploitation Linux Temps Réel, nommé RTAI (Realtime Application Interface). Ce système d'exploitation, comme son nom l'indique est un système dit temps réel, c'est à dire que ce système nous permet de garantir des temps de réponse à des périodes fixes dans le temps.

Nous utilisons ce système dans notre expérimentation, pour pouvoir simuler la présence d'autres systèmes temps réel sur l'ordinateur, en créant une tâche pour notre calcul de vitesse et d'autres tâche pour "parasiter" le bon déroulement du calcul.

1. Pourquoi travailler avec RTAI ?

Nous utilisons RTAI pour réaliser un ordonnancement de tâches temps réel qui nous permet de simuler la réalité des calculateurs embarqués.

Dans notre expérimentation, nous utilisons un ordonnancement à trois tâches, une tâche pour le calcul à proprement parlé et deux tâches pour occuper le processeur. Nous arrivons ainsi à faire varier la gigue et le retard entre l'entrée et la sortie. Nous pouvons observer les effets de ces perturbations sur la commande du moteur et le bon fonctionnement du système.

Le signal que nous recevons nous donne la tension de sortie du capteur, afin de réaliser le calcul de la commande, il nous faut la vitesse du moteur, nous effectuons donc une conversion tension-vitesse(rad.s^{-1}). Cette conversion s'effectue en multipliant la tension obtenue par un facteur d qui vaut $d = 0{,}64$.

2. Programmation

Pour clarifier le programme et simplifier les modifications, nous avons créé une bibliothèque qui rassemble toutes les variables globales, cette bibliothèque porte le nom de *constantes.h.*

Dans le module que nous créons, nous utilisons le calcul avec virgule flottante, pour obtenir des résultats plus précis. Nous ne pouvons pas utiliser la procédure *printk()* pour afficher les résultats de nos calculs car cette procédure ne gère pas l'affichage des flottants.

Le module comporte trois tâches, une tâche de calcul, *pid*, et deux tâches de charge processeur, *perte1* et *perte2*.

La tâche de calcul, *pid*, commence par l'acquisition de la valeur de la conversion analogique-numérique issue de l'appel à la procédure *readad()*. La valeur récupérée est ensuite transformée en vitesse. La transformation consiste en une multiplication par 2, car en sortie du capteur nous divisons par 2 la tension pour éviter les saturations, puis par une division par 409,5 pour obtenir une valeur de la tension comprise entre 0 et 10V. La tension obtenue est ensuite convertie en vitesse en divisant la tension par K_d .
On calcule ensuite la nouvelle commande à envoyer en utilisant la relation définie précédemment dans l'étude théorique :

$$u_k = u_{(k-1)} + K*(e_k - e_{(k-1)}) + (K_i * T_e * e_k)$$

avec e_k= *objectif – vitesse du moteur (en rad.*s^{-1}*)*, objectif étant la consigne, $e_{(k-1)}$ la valeur de l'erreur à la dernière période et $u_{(k-1)}$ la valeur de la commande précédente.
Avant d'envoyer la valeur de la nouvelle commande, on transforme la valeur de la commande, qui est en *rad.*s^{-1}, en une tension à envoyer au moteur en divisant par G_m.

Si la valeur que l'on souhaite envoyer dépasse 10V ou est inférieure à 0V, lors de la multiplication par 409,5 pour obtenir la valeur numérique correspondant à la tension en sortie du CNA de la carte, on peut obtenir une valeur supérieure à 4095 ou inférieure à 0, réalisant ainsi un « overflow », ou un « underflow ». Pour éviter ces problèmes, nous limitons la tension de sortie à la plage de tension 0-10V.

Une fois la limitation faite, on multiplie par 409,5 pour obtenir une valeur à convertir comprise entre 0 et 4095 pour obtenir en sortie une tension comprise entre 0 et 10V.

La valeur de la tension souhaitée est envoyée via l'appel à la procédure *setda()*.

Les tâches d'occupation de processeur, *perte1*, *perte2*, font appel à la procédure *perte_de_temps()* qui occupe le processeur avec une boucle *for*. Pendant que le processeur est dans la boucle *for*, rien ne se passe. La valeur de l'attente qui doit être faite est passée en paramètre et se compte en milisecondes.

H. Problèmes rencontrés

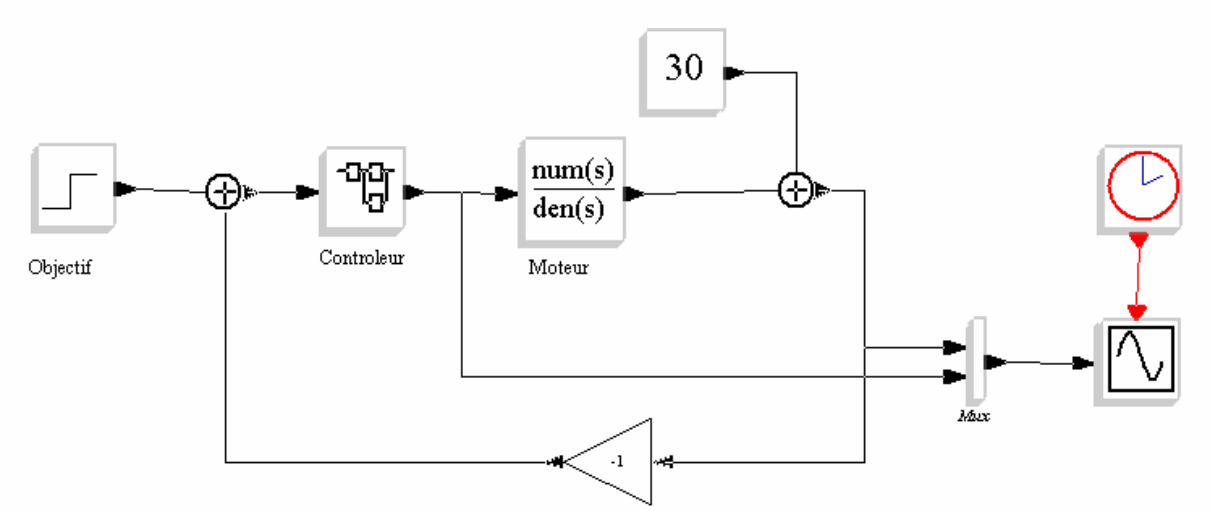
Dans notre première approche nous pensions effectuer les opérations de calcul dans un programme en C indépendant de RTAI et le reste, c'est à dire l'acquisition et l'envoi de la commande, sur le noyau RTAI. Nous utilisions pour cela des FIFOS (First In First Out, piles de stockage), mais nous créions ainsi des nouveaux temps d'attentes que nous ne pouvions pas maîtriser, donc nous ne répondions pas au sujet.

Dans une deuxième approche, nous avions choisis de créer une tâche par opération, c'est à dire, une tâche pour l'acquisition, une tâche pour le calcul et enfin une tâche pour l'envoi de la commande. Pour diffuser les données entre les tâches, nous utilisions une méthode d'envoi de donnée qui est disponible sous RTAI, mais du fait de l'utilisation de cette méthode, nous créions des dépendances entre les tâches, ce qui fait que nous ne pouvions pas voir le retard entre l'entrée et la sortie, puisque celui-ci était fixe.

IV. Comparaison entre la simulation et la réalité

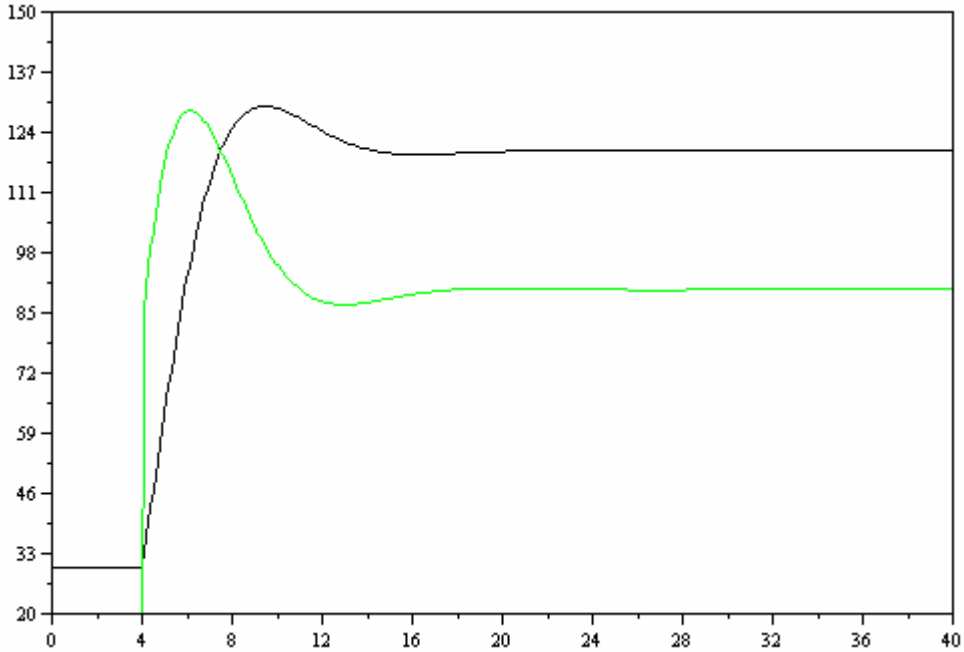
Pour pouvoir évaluer la qualité des simulations, nous avons comparé les résultats théoriques avec ceux trouvés en pratique. Cette comparaison porte sur les courbes et l'ordonnancement. Les expérimentations ont été faites sur deux jeux de tâches différents, jeu 1 et jeu 2, et sur deux commandes différentes, un échelon et une commande sinusoïdale.
Chaque jeu de tâche est testé en non préemptif et en préemptif pour voir l'influence de ces deux types d'ordonnancement sur les retards E/S.

Le schéma global de la simulation est le suivant :



La vitesse objectif est de 120 rad.s^{-1} et la vitesse initiale est de 30 rad.s^{-1}. On visualise à la fois la vitesse et la commande générée par le contrôleur.

En considérant un modèle sans aucun retard et avec une période d'échantillonnage de l'ordre de la nanoseconde, on obtient le résultat suivant en simulation :



Le maximum atteint est 130 rad.s^{-1} et le temps de stabilisation est de 16 secondes. En vert, on observe la commande qui est envoyée au moteur pour atteindre la vitesse consigne.

A. Étude des résultats avec le jeu 1

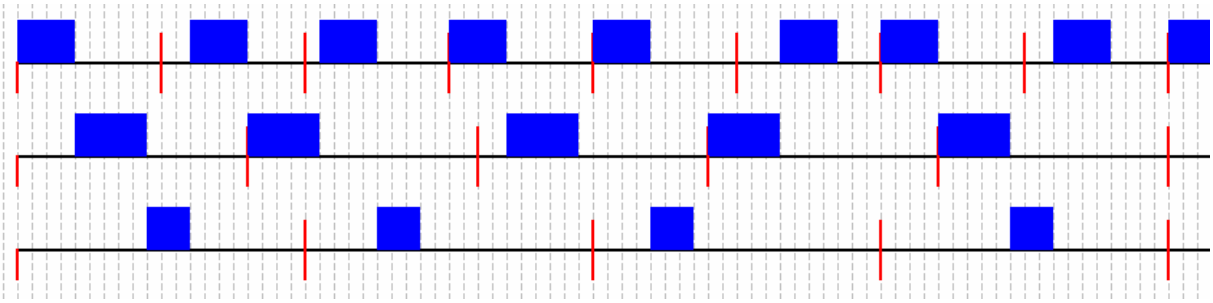
Jeu de tâches 1 :

	Période (ms)	Temps Tâche (ms)
T1, le PID	1000	150
T2, tâche vide	800	250
T3, tâche vide	500	200

L'ordonnancement utilisé est un rate monotonic (RM) que nous avons simulé grâce au programme java fourni. On obtient ainsi les valeurs de la gigue moyenne et les temps exacts de début et de fin des tâches.

1. Cas Non préemptif

Voici le chronogramme représentant l'ordonnancement des 3 tâches temps réels, où la dernière tâche correspond au PID. L'unité de temps est de 50 ms par barres verticales.



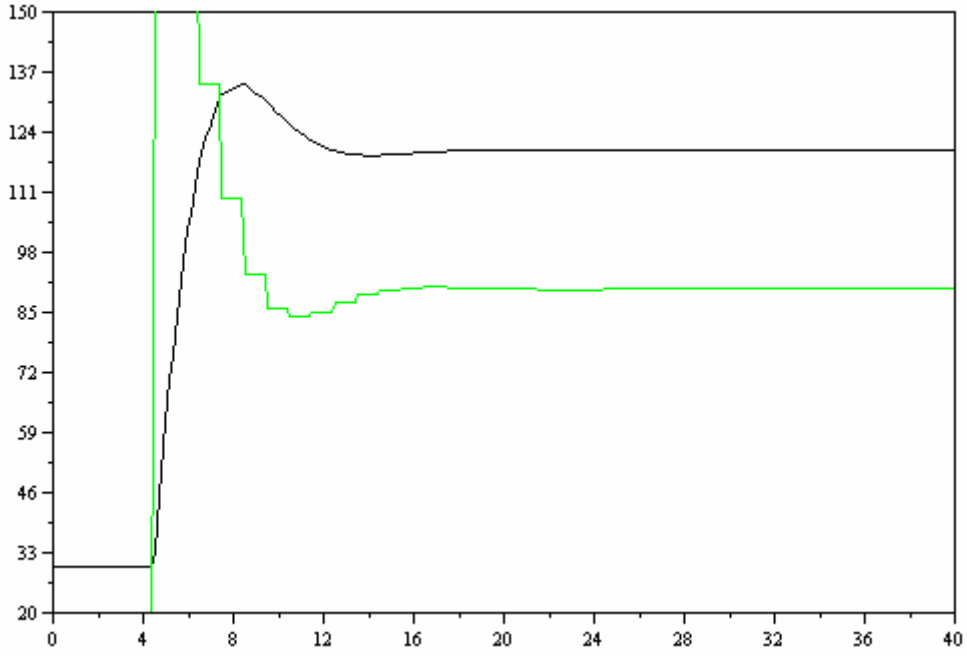
Ordonnancement des 3 tâches

- Modèles avec gigue et retards fixes

Lorsque l'on considère le jeu 1 en RM non préemptif, on obtient :

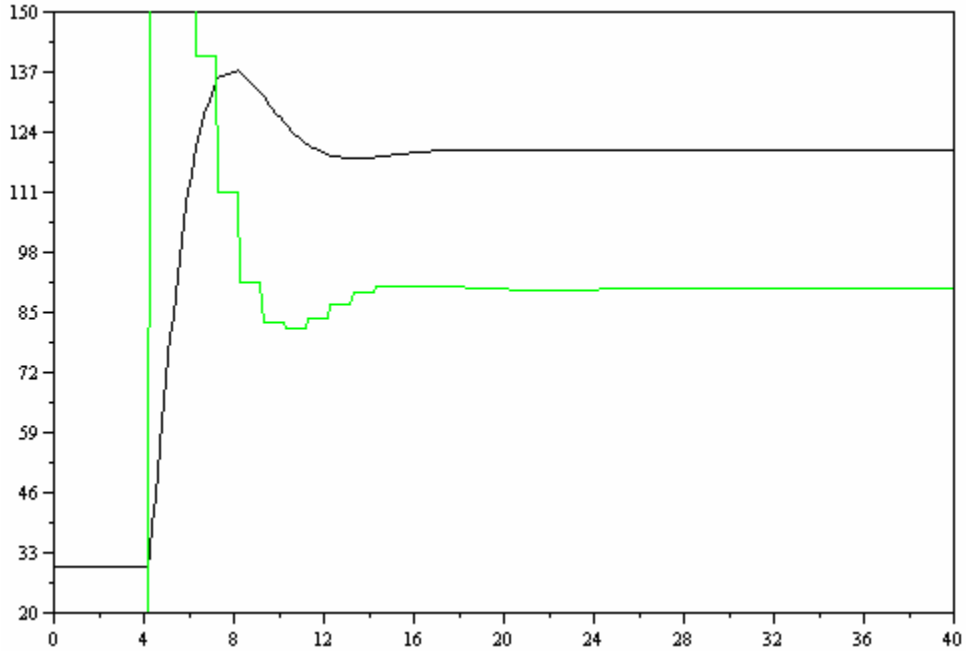
Gigue moyenne : 300 ms et Retard E/S moyen : 150 ms

En insérant ces valeurs dans le modèle de retards fixes, on obtient la courbe suivante :

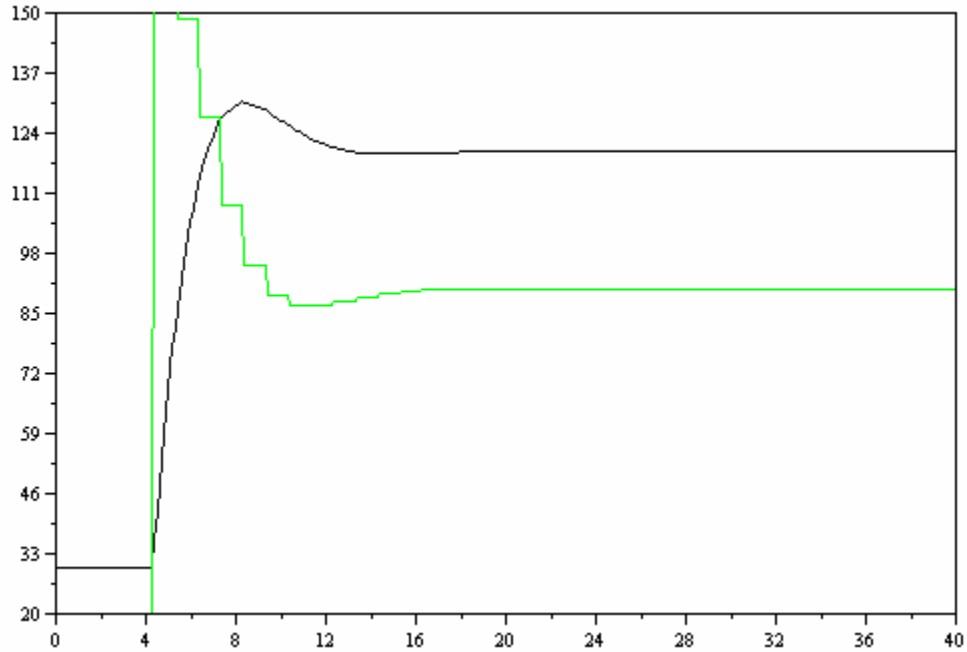


On obtient un maximum de 135 rad.s^{-1} et un temps de réponse d'environ 10 secondes. On peut déjà remarquer que l'insertion de retards fait varier la réponse du système.

On peut expérimenter l'importance de chacune des caractéristiques en séparant leur action.



Gigue nulle, Retard E/S fixe



Gigue fixe, Retard nul

On peut remarquer par rapport au modèle sans retard que l'allure de la courbe varie en fonction du retard E/S alors que la gigue fixe ne fait que retarder l'échantillonnage et, comme la valeur de la gigue est faible devant le temps de réponse du système, elle ne modifie pas l'allure générale de la courbe.

- Modèle avec gigue et retards aléatoire

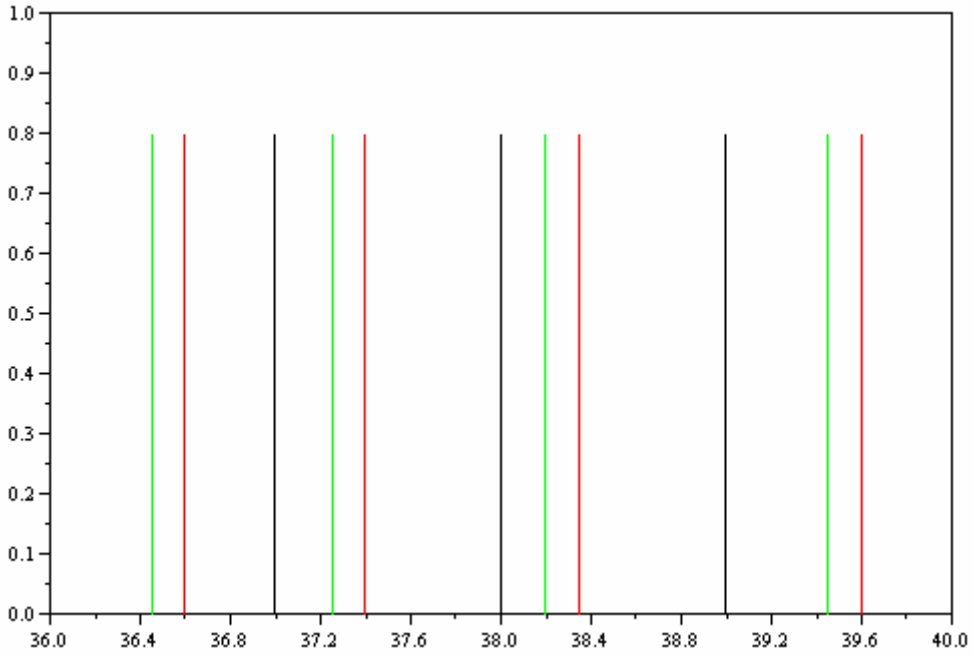
Ce modèle ne sera pas développé car les résultats obtenus sont quasiment identiques au modèle avec retards fixes. En effet, la variance des valeurs des retards n'est pas assez importante pour entraîner des différences

- Modèle avec gigue et retards exacts correspondant à l'ordonnancement théorique

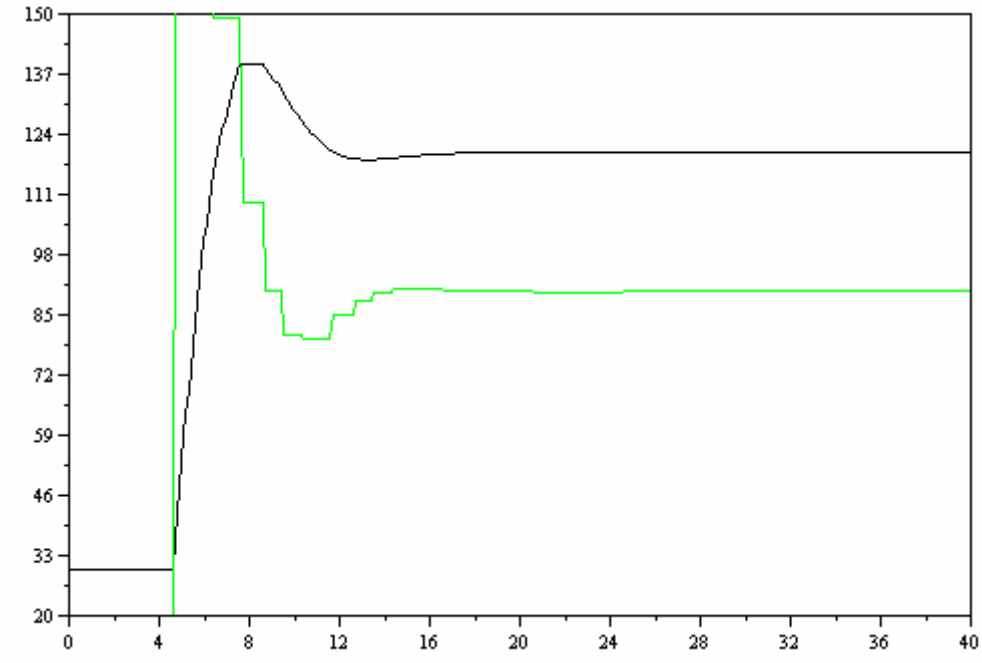
Grâce au logiciel d'ordonnancement java, on obtient les temps de gigue pour chaque période d'échantillonnage. Avec le jeu de tâches 1, on obtient un période absolue de 4.

Période	1	2	3	4
Gigue (ms)	450	250	200	450
Retards E/S (ms)	150	150	150	150

On peut visualiser l'ordonnancement en insérant ces valeurs dans le modèle décrit plus haut :



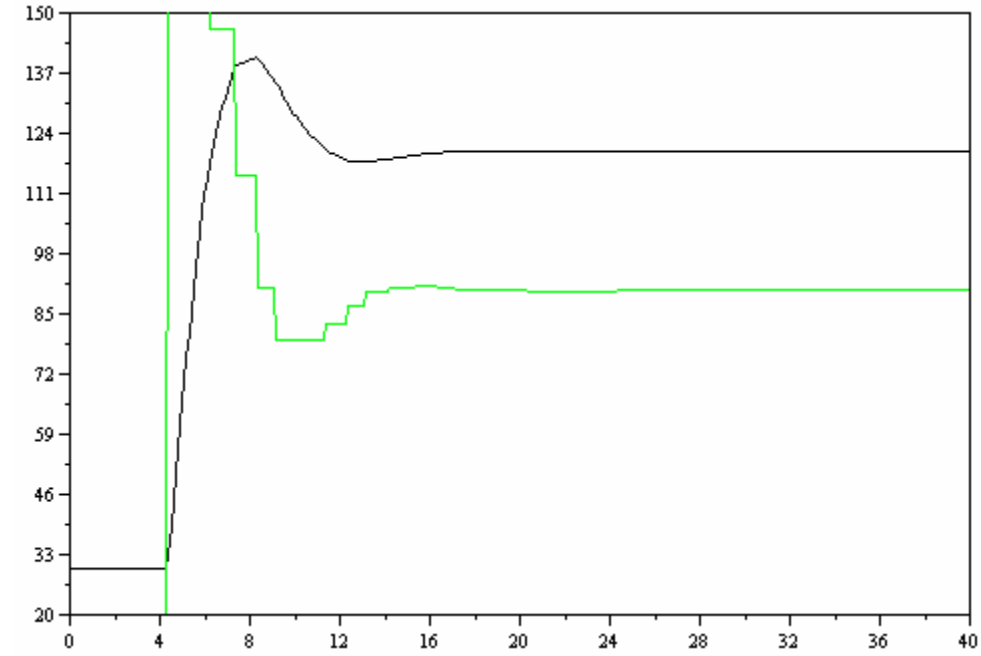
On peut voir que les temps de démarrage de la tâche varient en fonction de la période. Ceci influe sur la réponse du moteur comme on peut le voir ci-après.



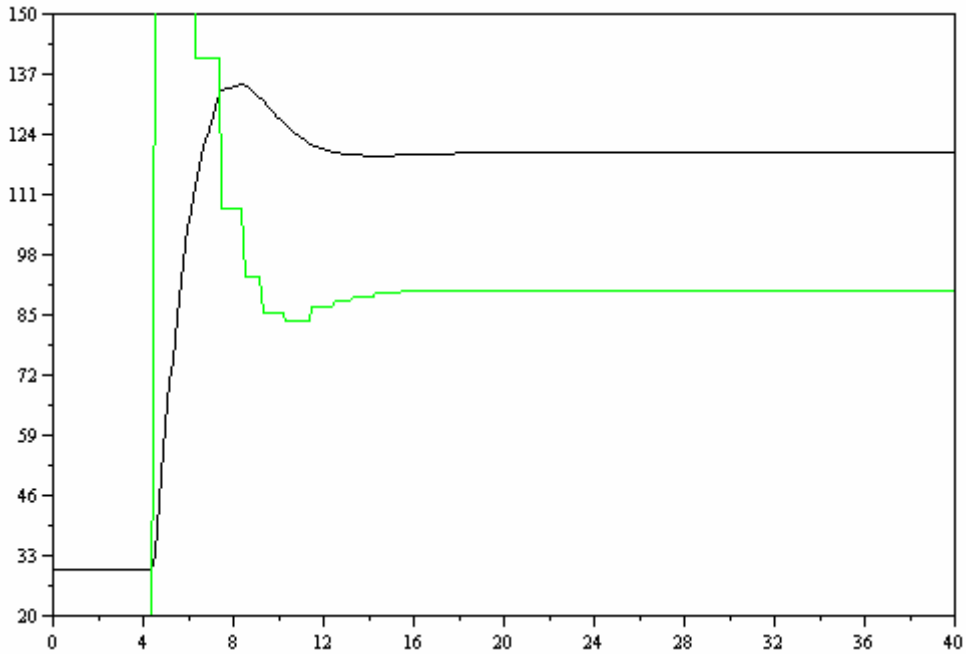
Courbe avec retards exacts

La valeur maximale atteinte est de 139 rad.s^{-1} et le temps de réponse est de 13 secondes.

Avec des valeurs de retards, on peut également tester l'importance de la gigue et du retard E/S variable.



Gigue nulle, Retard E/S variable

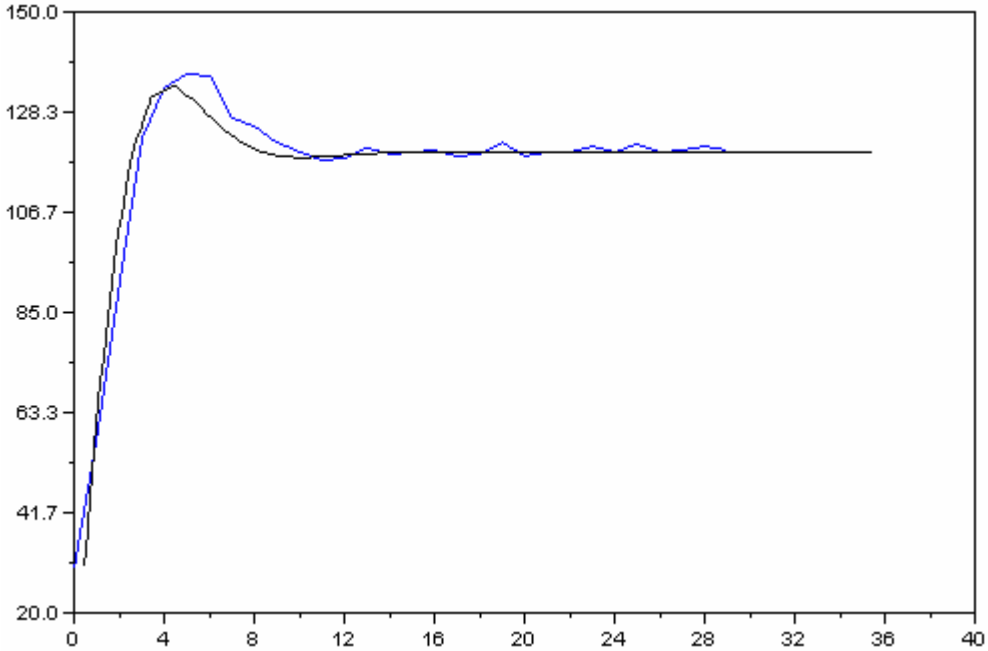


Gigue variable, Retard nul

Si on compare les trois courbes précédentes, on remarque que l'allure générale est respectée, mais que le modèle avec gigue nulle diffère plus des deux autres par la valeur de son maximum qui est de 136 rad.s^{-1} contre 140 rad.s^{-1} pour le modèle avec gigue nulle et 139 rad.s^{-1} pour la première courbe. Dans ce cas, la gigue variable ne semble pas avoir une importance aussi grande que le retard E/S.

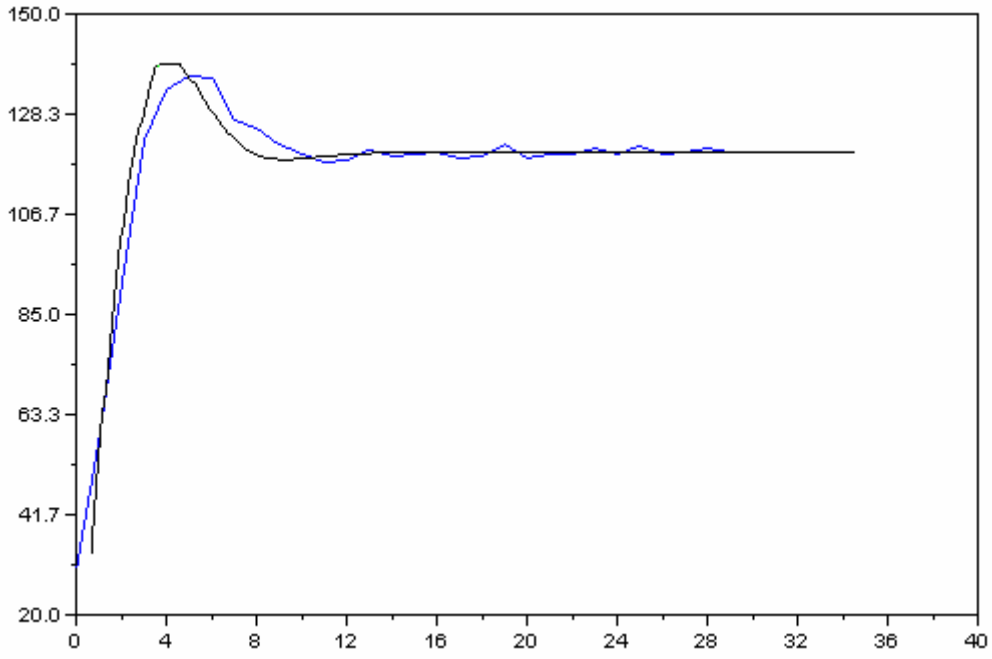
- Comparaison entre les résultats théoriques obtenus et les résultats de l'expérimentation.

On peut voir sur le schéma ci-dessous la différence entre la réponse réelle en bleu et la simulation avec retards fixe.



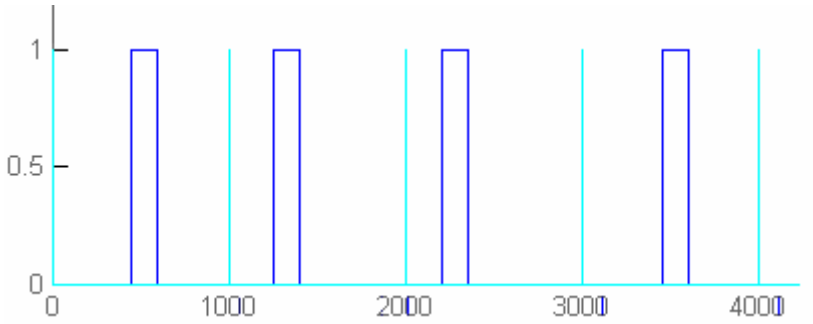
Courbe expérimentale Courbe théorique

La simulation est proche de la réalité mais la valeur maximale atteinte n'est pas tout à fait la même. On compare ensuite avec la courbe obtenue avec la simulation de l'ordonnancement réel et on obtient :



Courbe expérimentale Courbe théorique

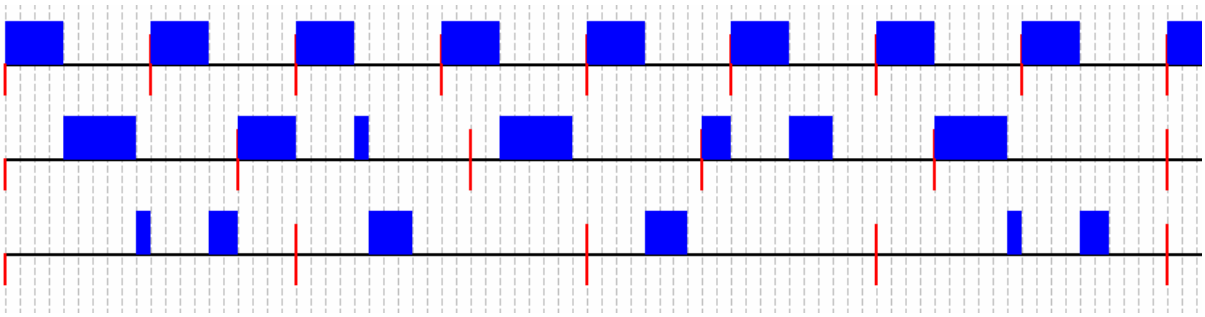
Sur cet exemple de jeu de tâche, la simulation n'est pas de meilleure qualité avec l'un ou l'autre modèle. On peut également comparer les ordonnancements théoriques et réels pour vérifier qu'ils sont identiques.
Avec les valeurs de temps récupérées par le programme RTAI, on obtient le chronogramme suivant :



Les valeurs correspondent à l'ordonnancement théorique. Il n'est donc pas la cause des différences entre la simulation et l'expérimentation.
Les causes de ces différences peuvent être dû à plusieurs paramètres :

- Les bruits de conversion. Un moteur est un système qui produit beaucoup de bruits parasites. Ainsi, les valeurs prises lors de la conversion varient.
- La précision de la conversion ; la valeur fournie par le convertisseur analogique numérique au PC est comprise entre 0 et 4095 (10 bits).
- La précision des paramètres du moteur. Le moteur possède un T_m caractéristique que nous avons calculé expérimentalement.

2. Cas Préemptif

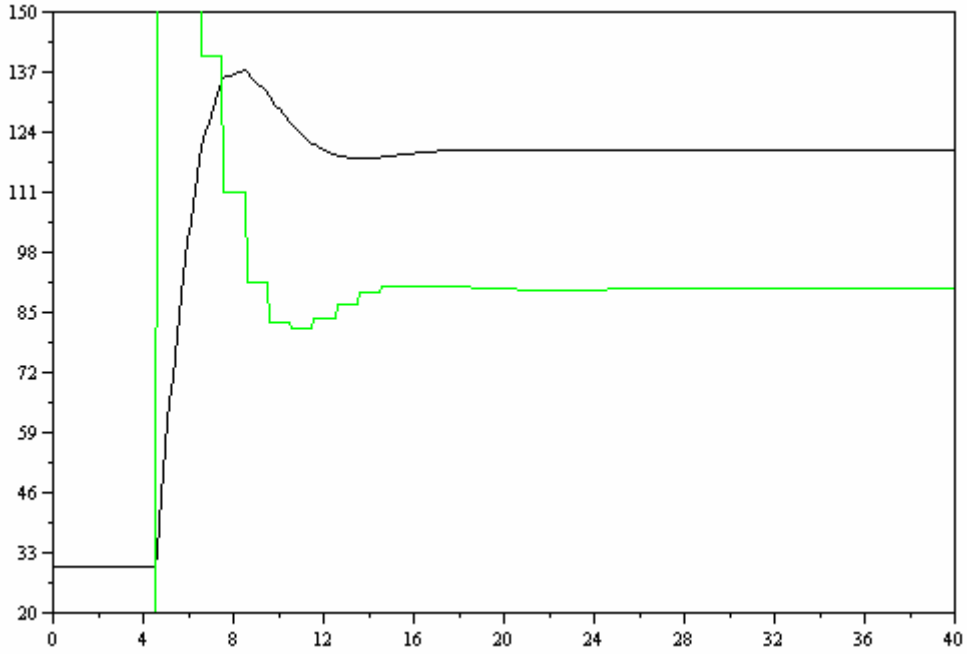


- Modèles avec gigue et retards fixes

Lorsque l'on considère le jeu 1 en RM préemptif, on obtient :

Gigue moyenne : 300 ms

Retard E/S moyen : 200 ms

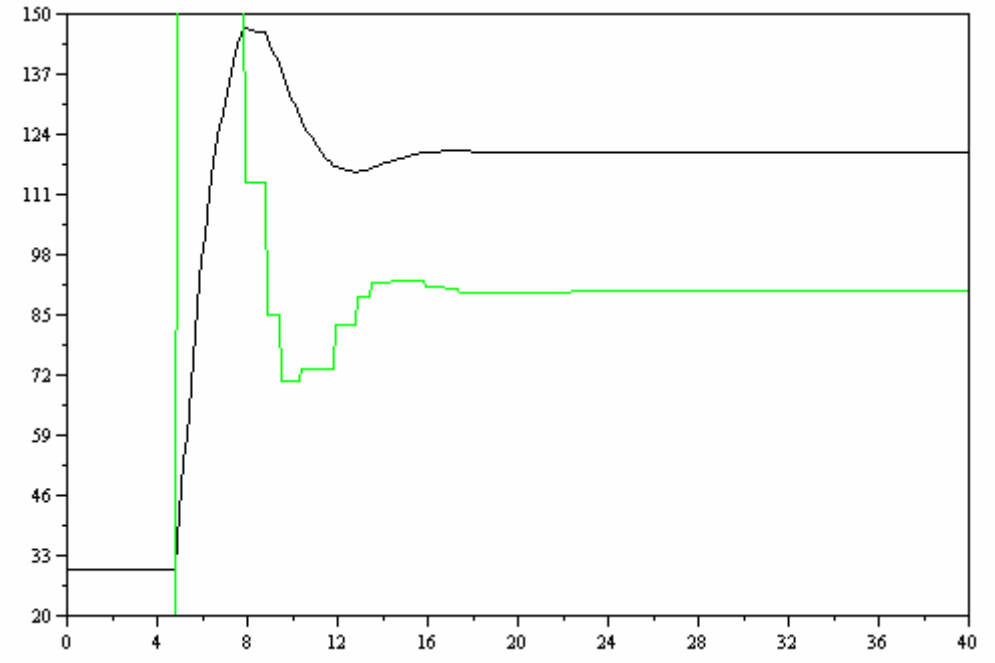


Valeur maximum : 138 rad.s^{-1}

- Modèle avec gigue et retards correspondant à l'ordonnancement

Avec le logiciel d'ordonnancement en java, on obtient les temps de gigue et de retards E/S :

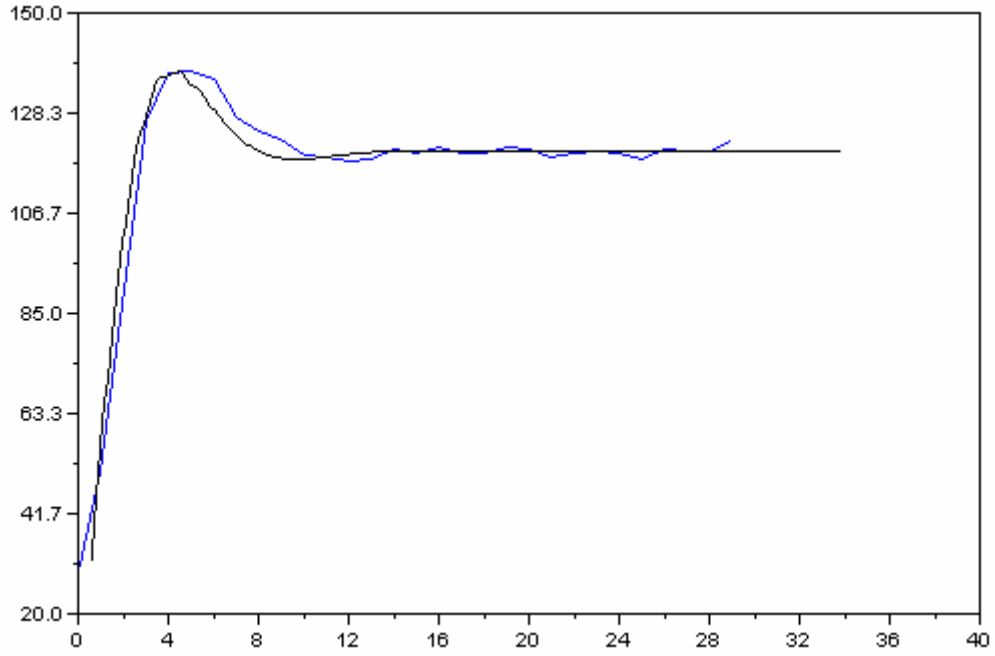
Période	1	2	3	4
Gigue (ms)	450	250	200	450
Retards E/S (ms)	350	150	150	350



Courbe avec retards exacts

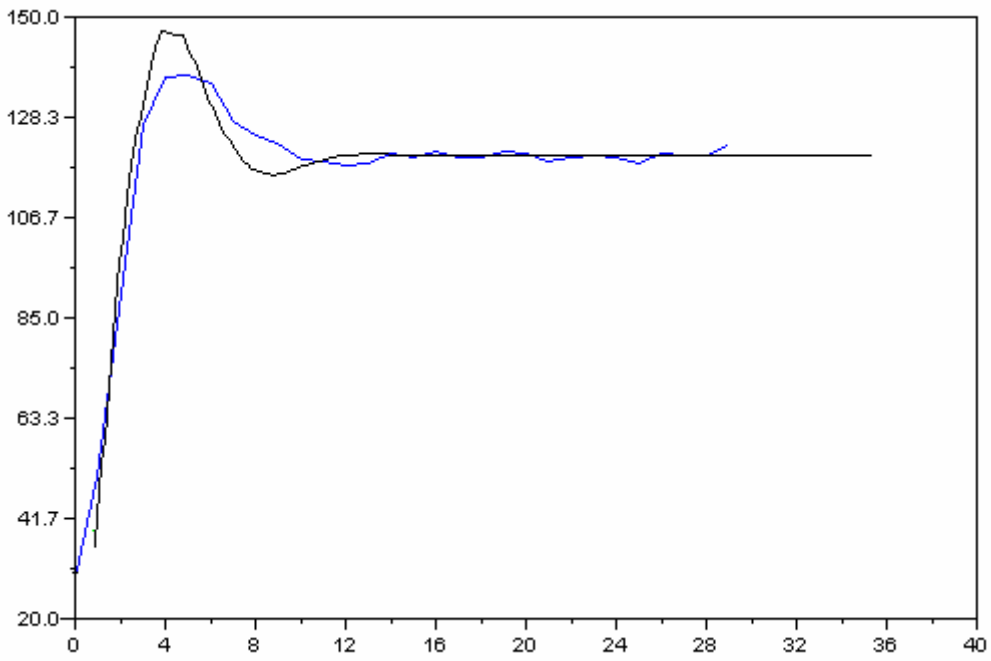
Valeur maximum : 145 rad.s^{-1}

- Comparaison entre les résultats théoriques obtenus et les résultats de l'expérimentation.



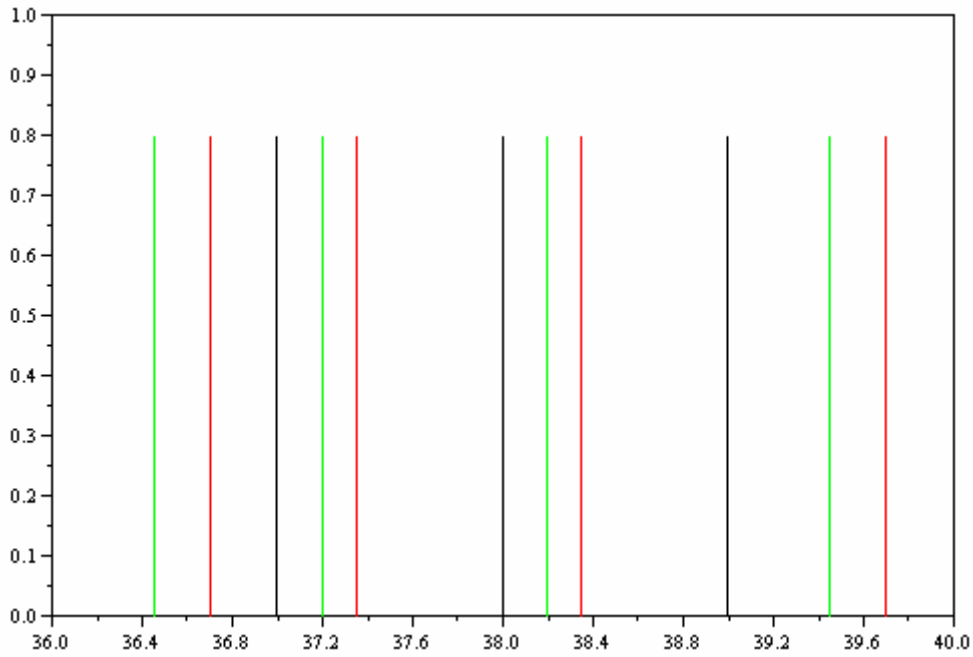
Courbe expérimentale Courbe théorique

La courbe obtenue en simulation en utilisant les valeurs des retards moyens est très proche de la courbe d'expérimentation. La valeur maximale atteinte et le temps de réponse sont les mêmes.

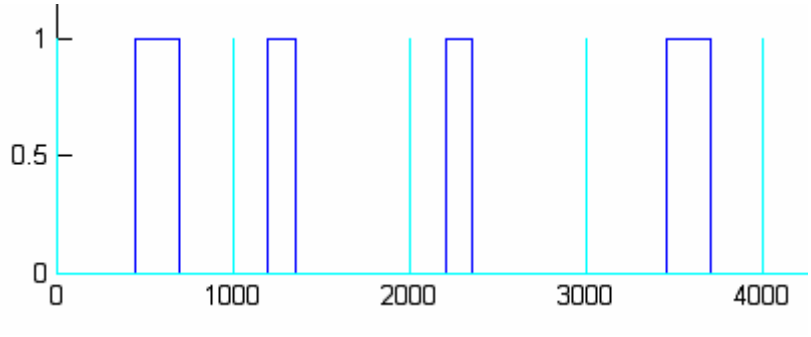


Courbe expérimentale Courbe théorique

En utilisant les valeurs de retard théorique, on obtient une simulation plus éloignée de la réalité. En observant l'ordonnancement réel, on peut voir qu'il y a des différences avec l'ordonnancement théorique.



Ordonnancement théorique

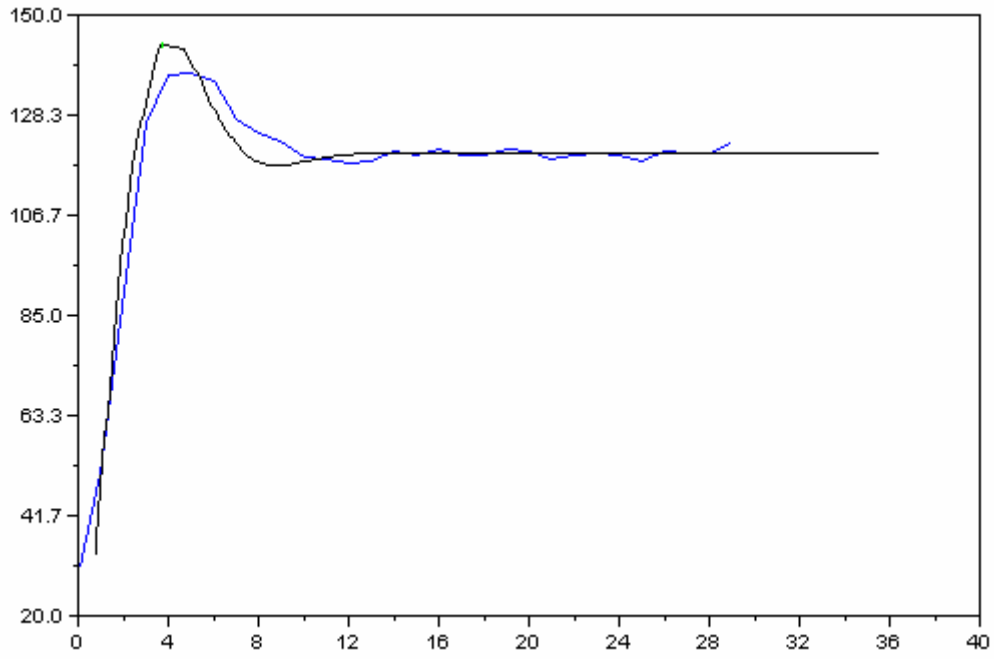


Ordonnancement réel

Période	1	2	3	4
Gigue théorique (ms)	450	250	200	450
Retards E/S théorique (ms)	350	150	150	350
Gigue expér. (ms)	450	200	200	450
Retards E/S expér. (ms)	250	150	150	250

Tableau des valeurs de retards

Ces différences peuvent être dus à la résolution de l'ordonnanceur théorique car l'unité de calcul que nous avons choisit est de 50 ms.
Si on réinjecte ces nouvelles valeurs dans la simulation, on obtient une courbe légèrement plus proche de la courbe d'expérimentation. Cependant le résultat n'est pas meilleur qu'avec des temps de retards fixe.



Courbe expérimentale Courbe théorique corrigée

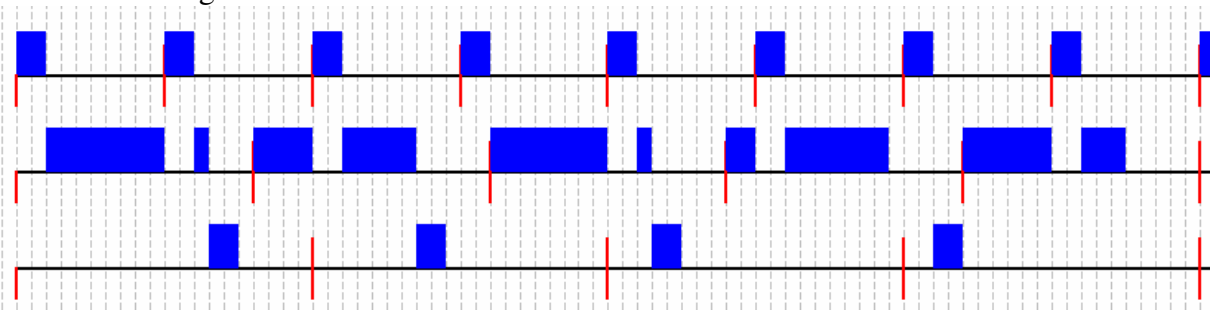
B. Étude des résultats avec le jeu 2

Pour pouvoir affiner l'analyse, nous avons refait les tests avec un deuxième jeu de tâches où la gigue varie de façon plus importante.

Jeu de tâches 2 :

	Période (ms)	Temps tâche (ms)
T1, le PID	1000	100
T2, tâche vide	800	450
T3, tâche vide	500	100

Voici le chronogramme de l'ordonnancement :



Avec ce jeu, le comportement de la tâche du PID ne change pas si l'ordonnancement est préemptif ou non préemptif, en théorie.

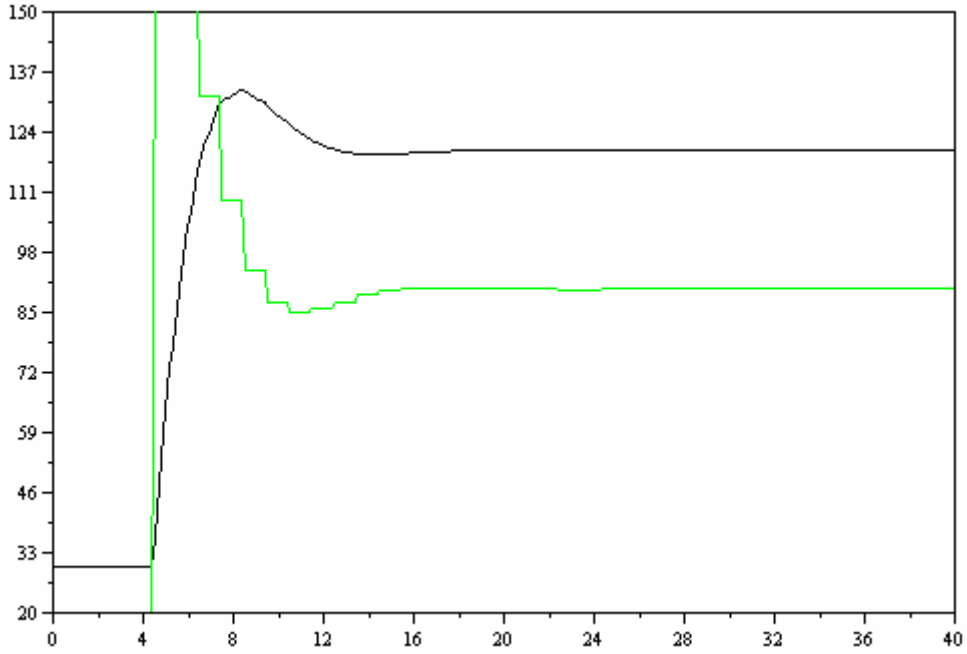
- Modèles avec gigue et retards fixes

Lorsque l'on considère le jeu 1 en RM préemptif, on obtient :

Gigue moyenne : 300 ms

Retard E/S moyen : 100 ms

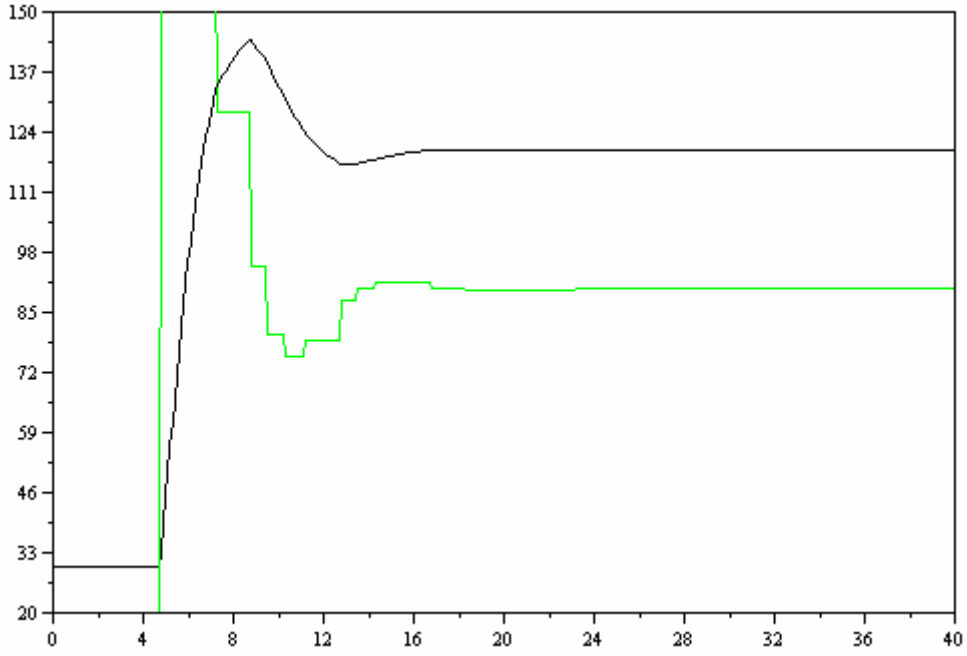
Voici la courbe obtenue avec le modèle :



Valeur maximum : 138 rad.s^{-1}

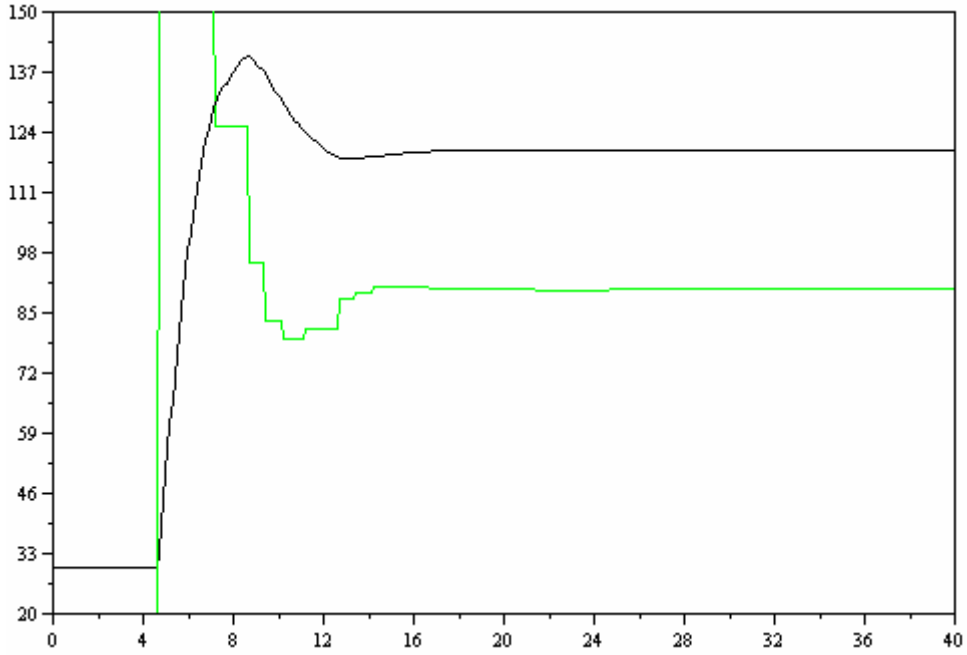
- Modèle avec gigue et retards correspondant à l'ordonnancement

Période	1	2	3	4
Gigue (ms)	650	350	150	100
Retards E/S (ms)	100	100	100	100

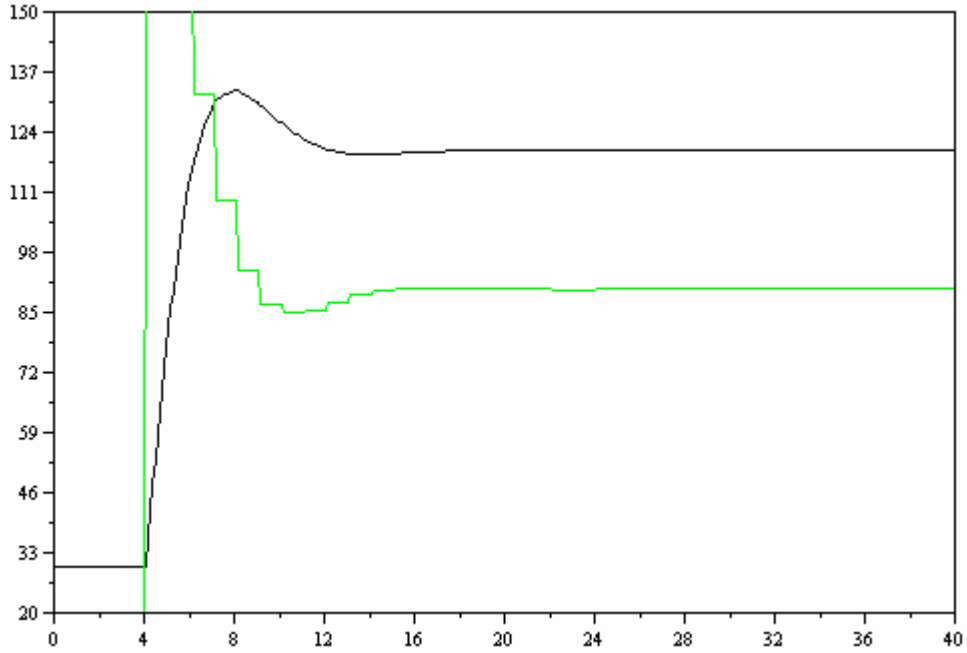


Courbe avec retards exacts

Valeur maximum : 144 rad.s^{-1}



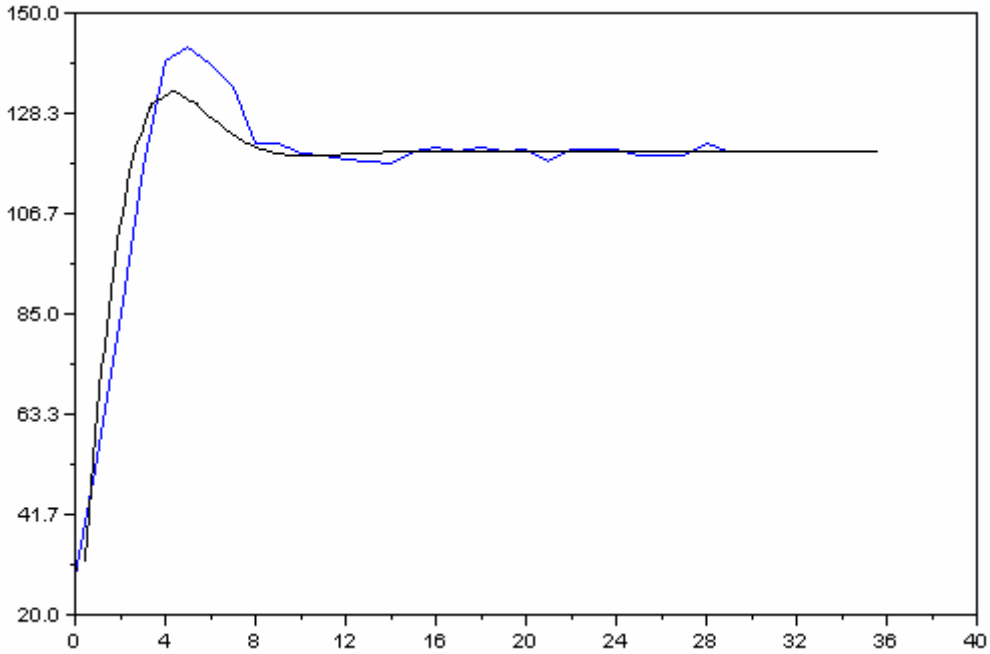
Gigue variable, Retard nul



Gigue nulle, retard variable

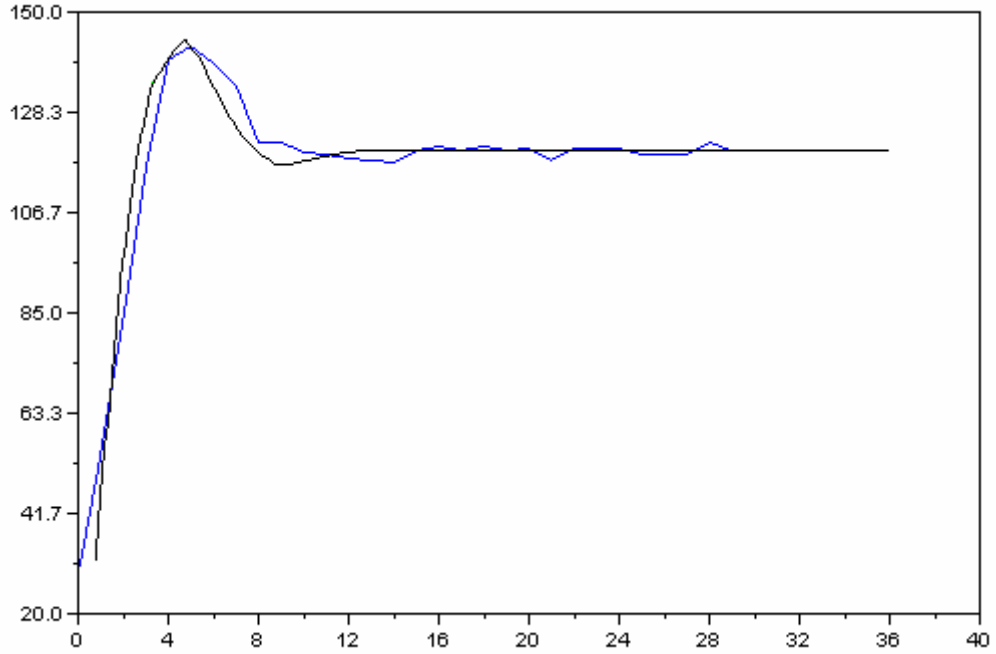
Dans cette exemple de jeu de tâches, on voit que la gigue agit plus sur l'allure de la courbe que le retard E/S.

- Comparaison entre les résultats théoriques obtenus et les résultats de l'expérimentation.



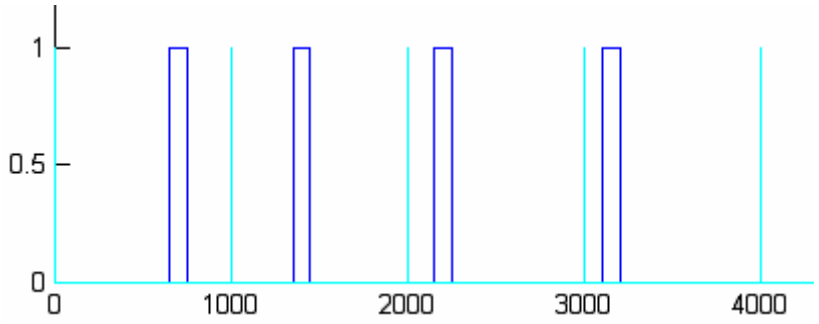
Courbe expérimentale Courbe théorique avec retards fixes

On peut voir sur la courbe précédente que le modèle avec retards fixes ne permet pas d'approcher correctement le comportement réel du système. Cependant, avec le modèle avec retard exact de l'ordonnancement, on obtient un résultat meilleur qu'avec le jeu de tâche 1.

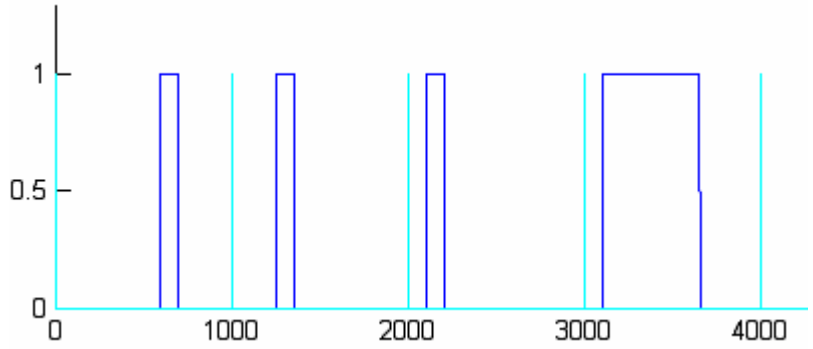


Courbe expérimentale en non préemptif Courbe théorique avec retards exacts

De plus, les ordonnancements théoriques et pratiques sont différents en préemptif.



Ordonnancement non préemptif réel

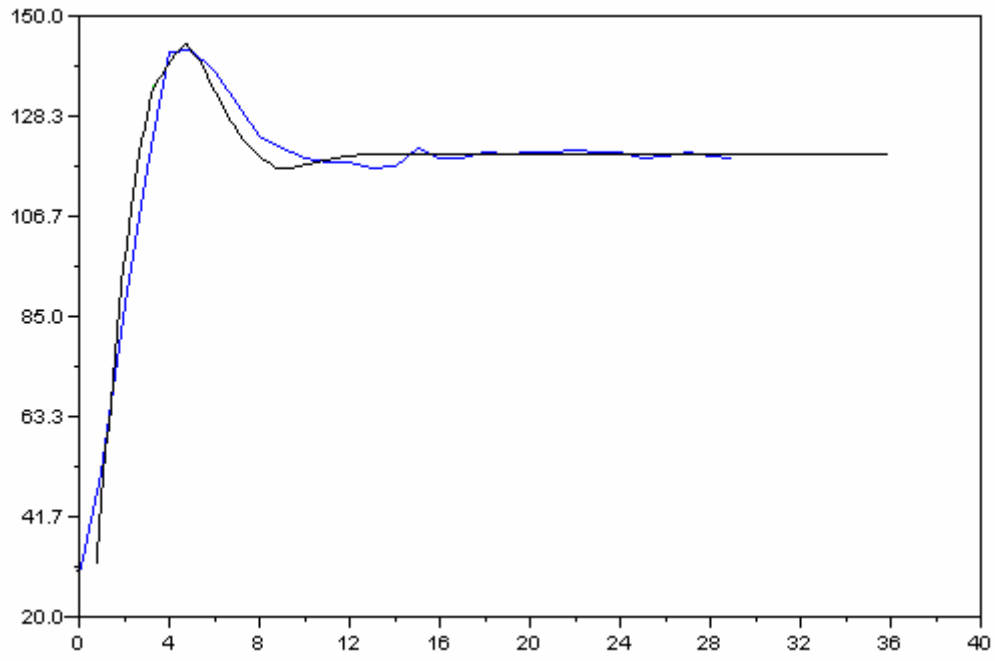


Ordonnancement préemptif réel

Période	1	2	3	4
Gigue théorique (ms)	650	350	150	100
Retards E/S théorique (ms)	350	150	150	350
Gigue expér. (ms)	600	350	100	100
Retards E/S expér. (ms)	100	100	100	550

Tableau des retards

Cependant, la réponse du système ne varie pas sensiblement. On a toujours un modèle proche de la réalité.



Courbe expérimentale en préemptif Courbe théorique avec retards exacts

L'étude avec la commande en sinus n'a pas été développée ici car elle n'apporte pas de précision supplémentaire. Les courbes sont disponibles en annexes.

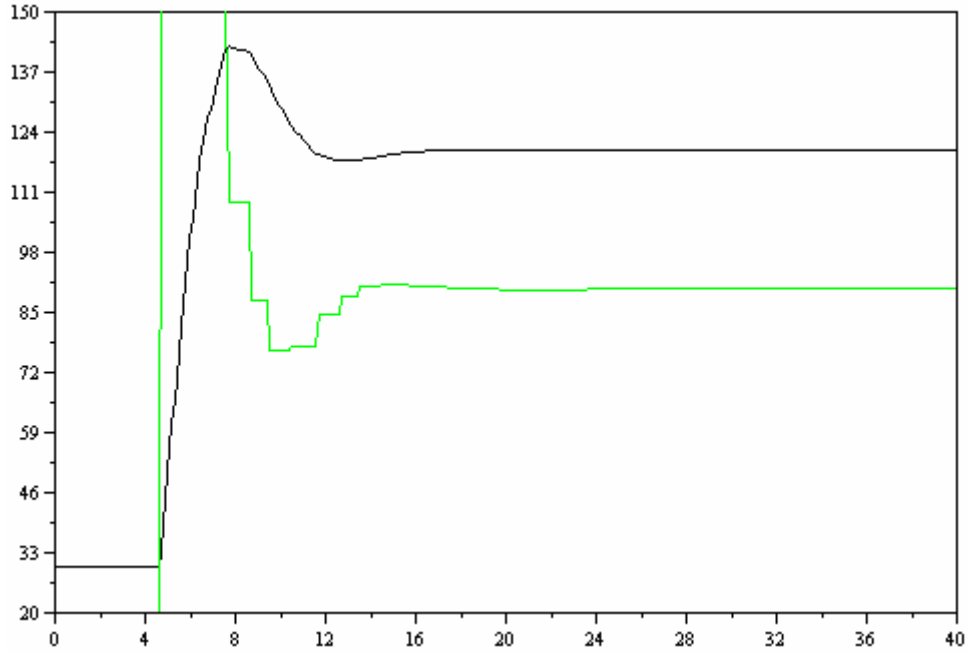
V. Conclusion sur les résultats

Malgré le petit échantillon de valeurs que nous avons pu tester, nous pouvons déjà tirer des conclusions sur notre expérimentation. Cependant ces conclusions seraient à vérifier avec un nombre de jeux de tâches beaucoup plus important et sur des systèmes beaucoup plus variés.

1. Le premier résultat que l'on peut tirer est que d'utiliser les valeurs de l'ordonnancement exact ne fournit pas obligatoirement la meilleure simulation comme on peut le voir sur le jeu de tâches 1.

2. D'après les résultats obtenus, les deux paramètres, la gigue et le retard E/S semblent tous les deux aussi importants dans la modification de la réponse. Cependant, l'un ou l'autre peuvent être négligé si leur valeur est faible par rapport à la réponse du système. Il faudrait déterminer les proportions pour chaque paramètre à partir desquels ils deviennent prépondérants dans la réaction du système.
- Il faut aussi prendre en considération l'importance des variations de ces paramètres. En effet, dans le jeu de tâche 1, la gigue a une variance de 110 ms et un écart maximum entre les gigues de 250 ms alors que dans le jeu de tâches 2, la gigue a une variance de 180 ms et un écart de 550 ms. De plus, on peut calculer le temps maximum qui sépare deux sorties de commande. Pour la tâche 1, le temps maximum est de 900 ms ce qui est inférieur à la période d'échantillonnage alors que pour la tâche 2, ce temps est de 1750 ms, lors du passage de la période 4 à la période 1. Cette dernière valeur dépasse largement la période d'échantillonnage. L'ensemble de ces observations permet de mettre en évidence que la gigue joue un rôle important dans le jeu de tâche 2 et c'est pourquoi il est préférable d'utiliser la modélisation exacte de l'ordonnancement.

- Dans le jeu 1, la meilleure simulation en préemptif est donnée par la modélisation des retards par les valeurs moyennes de la gigue et des retards E/S alors que la simulation avec les valeurs de l'ordonnancement exact donne un résultat de moindre qualité. D'après les tests effectués, la modélisation des retards par une valeur moyenne fixe donne de bons résultats, ce que l'on peut vérifier avec les cas non préemptif des jeux de tâches étudiés. Ainsi, en fusionnant les deux méthodes, c'est-à-dire en utilisant des valeurs spécifiques aux périodes pour la gigue et en prenant la valeur moyenne pour les retards E/S, il est possible d'obtenir une simulation plus proche de la réalité.
Par exemple, pour l'ordonnancement du jeu 1 en préemptif, on obtient effectivement une simulation de meilleure qualité.



Jeu 1, Préemptif, Gigue exacte, Retard moyen

VI. Programmes Additionnels

Le listing de ces programmes est disponible en annexes.

1. Calculus.c

Dans l'optique de visionner la vitesse du moteur au cours du temps sur un graphique et pour récupérer les temps de début et de fin de chaque tâche, nous avons écrit un programme nommé *calculus* pour traiter les informations envoyées dans les piles fifos.
Pour cela on commence par créer quatre fichiers. Le premier fichier contient des couples de points `(temps,vitesse)` qui permettent de tracer la vitesse en fonction du temps avec le logiciel xgraph ou avec scicos. Les trois autres fichiers contiennent les temps de début et de fin de chaque tâche. Le reste du programme est une boucle qui va lire les valeurs dans les fifos pour les écrire dans leurs fichiers respectifs.

La première fifo contient la vitesse du moteur en rad.s^{-1}. On écrit dans le fichier les couples (temps,vitesse) avec t = i*Te pour la i-ème vitesse reçue. On pourra alors comparer avec les courbes théoriques et déterminer si il y a une gigue ou un retard E/S. On obtient un fichier graphe_Tex.dat :

```
0.000000 29.473717
1.000000 56.786030
2.000000 91.859749
3.000000 122.708908
4.000000 133.515945
5.000000 136.758041
6.000000 135.972076
7.000000 127.031723
(...)
```

Les trois autres fifos contiennent les temps de début et de fin de chaque tâche, ils sont écrits dans les fichiers `tempo_x.txt` de la manière suivante:

```
d750
f1250
d1250
f1400
d2250
f2750
d3750
(...)
```

2. Parse.c

Une fois l'expérimentation terminée, on souhaite pouvoir observer le comportement de l'ordonnancement. Sous RTAI, il existe un logiciel qui permet de lire les traces d'exécution d'un ordonnancement, mais la version de ce logiciel sur l'ordinateur que nous utilisions n'étant ni stable ni utilisable, nous avons écrit un programme nommée *parse* pour créer à partir du fichier texte décrit ci-dessus (tempo_x.txt), une image de l'ordonnancement facilement exploitable.

Pour cela, nous parcourons le fichier, dont le format est celui de tempo_x.txt, pour en extraire les débuts et fins de la tâche. Nous nous basons sur le principe que seul les changements d'états nous

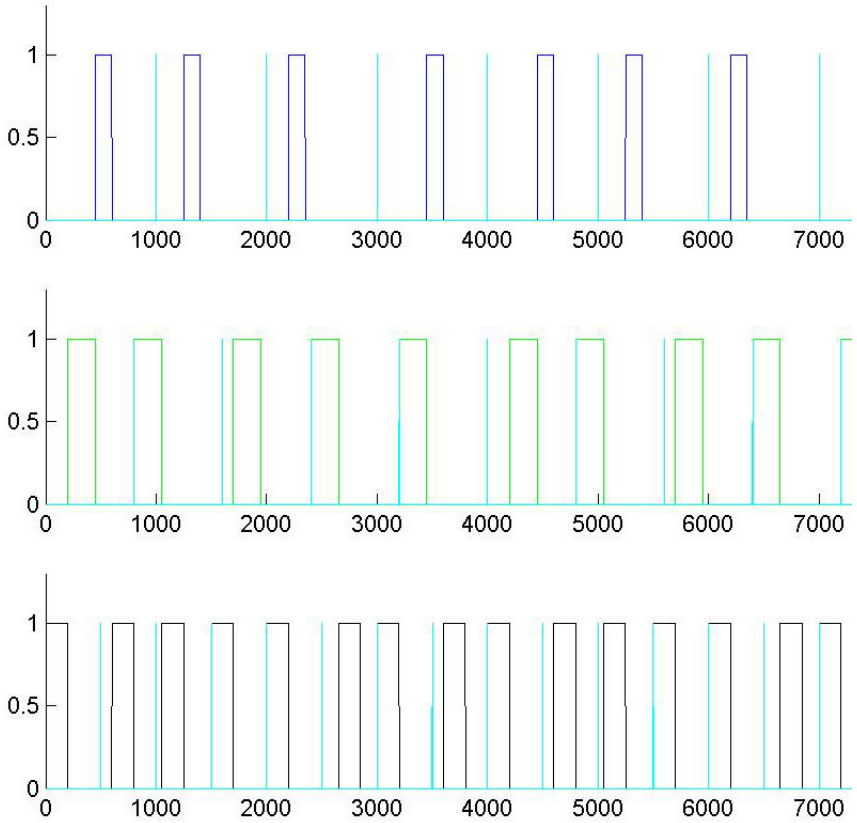
intéressent. Donc pour chaque changement d'état, nous plaçons deux points, un avant ayant pour valeur l'ancien état et un sur le changement d'état, avec la valeur du nouvel état.
Nous obtenons ainsi des fichiers de type chrono_x.dat, que l'on affiche avec matlab dont le format est le suivant :

```
449   0
450   1
599   1
600   0
1249  0
1250  1
1399  1
(...)
```

3. Script.c

Pour pouvoir afficher les chronogrammes correspondant à l'ordonnancement, nous avons utilisé le logiciel Matlab pour obtenir un affichage graphique de qualité.

Nous obtenons des chronogrammes de ce type :



On peut ainsi observer les temps d'activation et la durée de chaque tâche. Pour observer la préemption, il faut connaître la priorité des tâches et observer les temps de chevauchement de celles-ci

Conclusion

La complexité des systèmes embarqués actuels entraîne des temps de développement souvent très long et très coûteux. L'impératif de sécurité et de fiabilité nécessite de faire de nombreux tests pour obtenir un résultat qui répond au cahier de charges. Le rôle de l'automaticien est de fournir un modèle du procédé le plus fidèle et le plus rapidement possible, c'est-à-dire un modèle qui prend en compte les paramètres du système ainsi que les retards dû à l'implémentation du modèle sur un calculateur.
Notre projet a abordé ici une méthodologie permettant de remonter au plus tôt les retards dans le modèle de simulation. Cependant, l'étude des différents modèles décrits n'ont été expérimenté que sur un seul procédé, un asservissement de vitesse. Pour pouvoir les valider, il faudrait effectuer un grand nombre de tests avec plusieurs jeux de tâches différents et avec des procédés variables. Une étude statistique des résultats permettrait de déterminer l'importance de chacun des paramètres, mais le temps qui nous était imparti ne nous a pas permis d'aller plus lion dans notre démarche.

D'un point de vue pédagogique, ce projet nous a permis de nous familiariser avec l'ensemble des étapes du développement d'un système. Grâce à ce projet pluridisciplinaire, nous avons pu allié l'étude théorique et l'étude pratique pour concrétiser l'influence des retards sur la loi de commande. Nous avons appris à utiliser le logiciel de simulation Scilab Scicos ainsi que Matlab Simulink pour obtenir des courbes théoriques du comportement du moteur en vitesse que nous avons ensuite comparées avec les courbes expérimentales. Pour cela, il nous a fallu regrouper les connaissances acquises à l'ESIEE avec l'*étude des systèmes asservis* et sur la programmation RTAI pour pouvoir créer notre programme. De la modélisation jusqu'à l'implémentation, nous avons réussi à créer un système d'asservissement de vitesse qui fonctionne correctement cependant, nous n'avons pas pu faire tout ce qu'il faudrait faire pour un projet avec de telles conséquences sur les temps de développement.

Bibliographie

Concernant le temps réels et la théorie des retards :

- Rémy KOCIK et Yves SOREL, "De la modélisation à la réalisation : réduction du cycle de développement des applications temps réel distribuées".
http://www-rocq.inria.fr/syndex/pub/rts00/rts00.pdf

- "Vision théorique de l'AAA dans les systèmes temps réel distribués embarqués".
http://www-rocq.inria.fr/~cucu/AAAtheorique.ppt

- "On the Management of Latency in the Synthesis of Real-Time Signal Processing Systems from Processing Graphs", Ph.D. Dissertation, University of North Carolina at Chapel Hill, Department of Computer Science, 1998.
http://www.cse.unl.edu/~goddard/Papers/Dissertation/OneSidedThesis.ps

- "An Algorithm for Automatically Obtaining Distributed and Fault-Tolerant Static Schedules" DSN2003, Dependable Systems and Networks Conference, San Francisco, California, USA, June 2003.
http://www-rocq.inria.fr/syndex/pub/dsn03/dsn03.pdf

- "Real-time scheduling for systems with precedence, periodicity and latency constraints", RTS2002 10th conference on Real-time and embedded systems, Paris, Mar. 2002.
http://www-rocq.inria.fr/syndex/pub/rts02/rts02.pdf

Concernant le logiciel Scilab/Scicos :

- "Calcul numérique avec Scilab", INRIA
http://www.ac-grenoble.fr/ugine/site/logiciels_libres/rapport008.html

- "Getting Started with Scicos", Wolffdata Scilab Notes,
http://www.wolffdata.se

- Bruno Pinçon, "Introduction à Scilab, Institut Elie Cartan Nancy", E.S.I.A.L., Université Henry Poincaré

- "Scicos tutorial", INRIA
http://www-rocq.inria.fr/scilab/doc/scicos_html/tutorial.html

- Laurent Fallet, "Mini-manuel de Scilab"
http://asi.insa-rouen.fr/~lfallet/docs/logiciels/scilab/manuel_Scilab.pdf

Concernant le contrôle de systèmes :

- Prof R. Wynne, "Control systems and PID tutorials", Sheffield Hallam University
http://www.shu.ac.uk/schools/eng/teaching/rw/index.html

- Christophe Boniver, “Synthèse d’un robot aéro-propulsé”,
http://www.montefiore.ulg.ac.be/services/systems/boniver.pdf

- Daniel JAUME et Michel VERGÉ, “Le calculateur numérique pour la commande des processus”, Techniques de l’ingénieur,
http://www.techniques-ingenieur.fr/

- Alina BESANÇON-VODA Sylviane GENTIL, “Régulateurs PID analogiques et numériques”, Techniques de l’ingénieur,
http://www.techniques-ingenieur.fr/

- Dominique JACOB, “Applications de la commande PID”, Techniques de l’ingénieur,
http://www.techniques-ingenieur.fr/

ANNEXES

Bibliothèque *constantes.h* 48
Fichier *task.c* 49
Fichier *calculus.c* 54
Shell linux *txt2dat* 57
Fichier *parse.c* 58
Script Matlab *script.m* 60
Expérimentation avec commande en Sinus 62

Bibliothèque *constantes.h*

Donne les constantes du système et les variables globales

```
// Défini le nombre de boucles que doit effectuer le programme
#define GERMAINE  30

// Caractéristiques du Moteur
#define Ke  0.0382      // Valeur expérimentale = 1/Gm
#define Kd  0.0497      // Adaptation de la tension en vitesse
#define Gm  26.15       // Gain en vitesse expérimentale

// Constantes pour le calcul
#define K   0.9         // Gain de la Proportionnelle
#define Ki  0.9         // Gain de l'Intégrale
#define Kderiv    0.01  // Gain de la Dérivée

#define Te  1.0         // Période d'échantillonnage en seconde
// Période d'échantillonnage en nanosecondes
#define PERIODE   (int)((Te)*1000000000)

#define TICK_PERIOD 500000    //  Durée entre deux tops d'horloge 500 µs


// Vitesse a atteindre en rad.s-1
#define objectif  120.0
// Jeu de tâches
#define D1  150         // Durée de la tâche 1
#define D2  250         // Durée de la tâche 2
#define D3  250         // Durée de la tâche 3

// Définit si l'ordonnancement est préemptif ou non; 1=oui, 0=non
#define preempt   1

// Définition du type u16 défini en RTAI
#define u16 unsigned short

// Définition des procédures lié à la carte d'acquisition
int init1710(void);
void setda(unsigned channel, unsigned value);
void setchanel(int in_channel);
u16 readad(void);
```

Fichier *task.c*

Créé un module à charger sur le noyau RTAI pour faire l'asservissement du moteur.

```
#include <linux/module.h>
MODULE_LICENSE("GPL");
#include <asm/io.h>

#include <asm/rtai.h>
#include <rtai_sched.h>
#include <rtai_fifos.h>
#include "constantes.h"

#define NUMERO      1

#define PRIORITE1    1

#define STACK_SIZE  2000

void perte_de_temps(int val);

static RTIME depart, depart_calc;;
static RT_TASK perte1,perte2,pid;
static int fifo1, fifo2, fifo3, fifo4;


//-------------------------------------------------------------------------------
// PID
//-------------------------------------------------------------------------------

void pid_code(int arg)
{

      int time_ms_1;
      static int boucle = GERMAINE;
      // Variables du PID
      float uk, uk_1, ek, ek_1, ek_2;
      float rad_s,val_f,tension,volt_commande;
      u16 resultat,commande_val;


      //Initialisation du calcul de la commande
      ek=0;
      ek_1=0;
      ek_2=0;
      uk=0;
      uk_1=0;

      while(boucle --)
      {

            // Envoi la valeur du temps au démarrage de la tâche via la FIFO 2
            time_ms_1=(rt_get_time_ns()-depart_calc)/1000000.0;
            rtf_put(2,&time_ms_1,sizeof(int));


            // Procédure donnant la valeur du capteur de vitesse
            resultat=readad();

            // Conversion du résultat de type u16 en float pour améliorer la
            // précision de calcul
```

```
            val_f = (float)resultat;

            // On multiplie par deux la valeur récupérée car la sortie
            // du capteur est divisée par 2 pour éviter les saturations
            // Conversion de la valeur du capteur en tension en volts
            tension = (val_f / 4096.0) * 10.0 * 2.0;


            // Transforme la tension en rad.s-1
            rad_s=tension/Kd;

            //Envoie la vitesse pour l'affichage et la génération d'un graphique
            // représentant la vitesse en fonction du temps
            rtf_put(1,&rad_s,sizeof(float));

            // Calcul de la commande a renvoyer
            // RTAI n'affiche pas les flottants

            // Calcul de l'erreur la commande et l'objectif
            ek=objectif-rad_s;
            // Calcul de la nouvelle commande à envoyer
            uk=uk_1 + K*(ek - ek_1) + (Ki * Te * ek);

            // Mise à jour des variables
            ek_2=ek_1;
            ek_1=ek;
            uk_1=uk;

            // Calcul de la tension à envoyer
            volt_commande = uk/Gm;
            // Limitation de la tension à envoyer
            if (volt_commande > 10.0)
                  volt_commande = 10.0;
            if (volt_commande < 0.0)
                  volt_commande = 0.0;

            // Conversion de la tension en une valeur numérique comprise entre 0
            // et 4095, au format u16
            commande_val = (u16)(volt_commande * 409.50);

            // Procédure pour faire durer la tâche
            perte_de_temps(D1);

            // Envoi de la donnée vers la carte d'acquisition sur le Port DAO_0
            setda(0,commande_val);

            // Envoi la valeur du temps à la fin de la tâche via la FIFO 2
            time_ms_1=(rt_get_time_ns()-depart_calc)/1000000.0;
            rtf_put(2,&time_ms_1,sizeof(int));

            // Attente de la prochaine activation de la tâche
            rt_task_wait_period();
      }
}

//-----------------------------------------------------------------------------
// Première perte de temps
//-----------------------------------------------------------------------------

void perte1_code(int arg)
{
```

```
	static int boucle = GERMAINE ;
	int time_ms_2;

	while(boucle --)
	{
		// Envoi le temps de début de tâche via la FIFO 3
		time_ms_2=(rt_get_time_ns()-depart_calc)/1000000.0;
		rtf_put(3,&time_ms_2,sizeof(int));

		perte_de_temps(D2);

		// Envoi le temps de fin de tâche via la FIFO 3
		time_ms_2=(rt_get_time_ns()-depart_calc)/1000000.0;
		rtf_put(3,&time_ms_2,sizeof(int));

		rt_task_wait_period();
	}
}


//-------------------------------------------------------------------------------
// Seconde perte de temps
//-------------------------------------------------------------------------------

void perte2_code(int arg)
{
	static int boucle = GERMAINE ;
	int time_ms_3;

	while(boucle --)
	{
		// Envoi le temps de début de tâche via la FIFO 4
		time_ms_3=(rt_get_time_ns()-depart_calc)/1000000.0;
		rtf_put(4,&time_ms_3,sizeof(int));

		perte_de_temps(D3);

		// Envoi le temps de fin de tâche via la FIFO 4
		time_ms_3=(rt_get_time_ns()-depart_calc)/1000000.0;
		rtf_put(4,&time_ms_3,sizeof(int));

		rt_task_wait_period();
	}
}

//-------------------------------------------------------------------------------
// Ralentissement de l'activité
//-------------------------------------------------------------------------------

// La paramètre val donne le nombre de miliseconde que doit durer l'attente
void perte_de_temps(int val)
{
	RTIME temps;

	temps = rt_get_time_ns();
	// Attente de val * 1 miliseconde
	while ( (rt_get_time_ns()-temps)< (val*1000000));
}

//-------------------------------------------------------------------------------
// Init !
```

```
//-----------------------------------------------------------------------------

int init_module(void) {

  int ierr;

  rt_set_oneshot_mode();

  // Choix du mode préemptif ou non
  rt_preempt_always(preempt);

  // On indique qu'on effectue des tâches avec virgule flottante grâce à
  // l'avant-dernier paramètre de la procédure rt_task_init
  ierr = rt_task_init(&pid, pid_code, NUMERO+1, STACK_SIZE, PRIORITE1+2, 1,0);
  ierr = rt_task_init(&perte1, perte1_code, NUMERO,STACK_SIZE,PRIORITE1+1, 1,0);
  ierr = rt_task_init(&perte2, perte2_code, NUMERO+2,STACK_SIZE,PRIORITE1, 1,0);

  printk("[tâche1 %d] cree code retour %d par programme %s\n",
NUMERO,ierr,__FILE__);
  printk("PERIODE=%d\n",PERIODE);

  if (!init1710()) // Teste si la carte est bien initialisée
  {
      printk("Carta gatova :) !\n"); // On affiche la carte est prête
      if (!ierr)
      {


            // Lancement du timer
            start_rt_timer(nano2count(TICK_PERIOD));

            // Crétion des FIFOS
            fifo1 = rtf_create(1,200);
            fifo2 = rtf_create(2,200);
            fifo3 = rtf_create(3,200);
            fifo4 = rtf_create(4,200);

            // Définition du canal de conversion du CAN
            setchanel(0);

            // Récupération du temps de départ
            depart = rt_get_time();
            depart_calc = count2nano(depart);

            // Lancement des tâches
            rt_task_make_periodic(&perte2,depart,nano2count(0.5*PERIODE));
            rt_task_make_periodic(&perte1,depart,nano2count(0.8*PERIODE));
            rt_task_make_periodic(&pid,depart,nano2count(1.0*PERIODE));
      }
      return ierr;
  }
  else
  {
      printk("Marche pas!!!!\n");
  }
  return 0;
}

//-----------------------------------------------------------------------------
// Nettoyage des modules
//-----------------------------------------------------------------------------
```

```
void cleanup_module(void)
{
  stop_rt_timer();
  setda(0,0);

  rtf_destroy(1);
  rtf_destroy(2);
  rtf_destroy(3);
  rtf_destroy(4);
  rt_task_delete(&perte1);
  rt_task_delete(&pid);
  rt_task_delete(&perte2);

}
```

Fichier *calculus.c*

Récupère les temps de début et de fin des tâches et récupère la vitesse du moteur

```
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include "constantes.h"


int main() {
	// fifo1 = temps reel -> programme C >> Vitesse
	// fifo2 = temps reel -> programme C >> Temps de la tâche 1
	// fifo3 = temps reel -> programme C >> Temps de la tâche 2
	// fifo4 = temps reel -> programme C >> Temps de la tâche 3
	int fifo1,fifo2,fifo3,fifo4;
	int i,j,k, count;
	float rad_s;
	int temps1,temps2,temps3;
	float moy_minimax, moy;

	// Fichiers pour tracer ensuite des graphiques
	FILE* fichier;
	FILE* fichier_temps_1;
	FILE* fichier_temps_2;
	FILE* fichier_temps_3;

	char buf_vit[50],buf_temps[50],filename[50];
	float t = 0;

	// Le nom du fichier dépend du Te qu'on utilise
	sprintf(filename,"graphe_Te%.3f.dat",Te);
	fichier = fopen(filename,"w");

	sprintf(filename,"tempo_1.txt");
	fichier_temps_1 = fopen(filename,"w");

	sprintf(filename,"tempo_2.txt");
	fichier_temps_2 = fopen(filename,"w");

	sprintf(filename,"tempo_3.txt");
	fichier_temps_3 = fopen(filename,"w");

	printf("Te = %f, Periode = %d\n",Te,PERIODE);

	// Test des ouvertures des Fifos pour la lecture des informations
	if( (fifo1 = open("/dev/rtf1",O_RDONLY)) < 0 )
	{
		printf("Erreur !");
		exit(-1);
	}
	else printf("Fifo1 ouverte\n");

	if( (fifo2 = open("/dev/rtf2",O_RDONLY)) < 0 )
	{
		printf("Erreur !");
		exit(-1);
	}
	else printf("Fifo2 ouverte\n");
```

```
if( (fifo3 = open("/dev/rtf3",O_RDONLY)) < 0 )
{
      printf("Erreur !");
      exit(-1);
}
else printf("Fifo3 ouverte\n");

if( (fifo4 = open("/dev/rtf4",O_RDONLY)) < 0 )
{
      printf("Erreur !");
      exit(-1);
}
else printf("Fifo4 ouverte\n");

i=0;j=0;k=0;
// On effectue le même nombre d'itération que dans l'ordonnancement
count = GERMAINE;

do {
      // Récupération de la vitesse du moteur
      read(fifo1,&rad_s,sizeof(float));
      // On envoi la date et la vitesse du moteur
      sprintf(buf_vit,"%f %f\n\0",t,rad_s);
      fwrite(&buf_vit,sizeof(char),strlen(buf_vit),fichier);
      t += Te;

      // Affichage dans le terminal de la vitesse
      // et de la valeur du compteur
      printf("i = %d\tRad/s:%.2f\n",count,rad_s);
      fflush(NULL);

      // Récupération des temps de début et de fin des tâches
      read(fifo2,&temps1,sizeof(int));
      // Si i==0, on est au début de la tâche, si i==1,
      //on est à la fin de la tâche
      if ((i%2)==0)
      {
            // Le 'd' indique la date de début de la tâche
            sprintf(buf_temps,"d%d\n\0",temps1);
            printf("d%d\n\0",temps1);
            fwrite(&buf_temps,sizeof(char),strlen(buf_temps),fichier_temps_1);
            i++;
      }
      else
      {
            // Le 'f' indique la date de fin de la tâche
            sprintf(buf_temps,"f%d\n\0",temps1);
            printf("f%d\n\0",temps1);
            fwrite(&buf_temps,sizeof(char),strlen(buf_temps),fichier_temps_1);
            i++;
      }

      read(fifo3,&temps2,sizeof(int));
      if ((j%2)==0)
      {
            sprintf(buf_temps,"d%d\n\0",temps2);
            fwrite(&buf_temps,sizeof(char),strlen(buf_temps),fichier_temps_2);
            j++;
      }
      else
      {
```

```
                sprintf(buf_temps,"f%d\n\0",temps2);
                fwrite(&buf_temps,sizeof(char),strlen(buf_temps),fichier_temps_2);
                j++;
            }

            read(fifo4,&temps3,sizeof(int));
            if ((k%2)==0)
            {
                sprintf(buf_temps,"d%d\n\0",temps3);
                fwrite(&buf_temps,sizeof(char),strlen(buf_temps),fichier_temps_3);
                k++;
            }
            else
            {
                sprintf(buf_temps,"f%d\n\0",temps3);
                fwrite(&buf_temps,sizeof(char),strlen(buf_temps),fichier_temps_3);
                k++;
            }

            count --;
        } while (count>0);

        // Une fois le décompte terminé, on ferme proprement les Fifos
        close(fifo1);
        close(fifo2);
        close(fifo3);
        close(fifo4);
        return 0;
}
```

Shell linux ***txt2dat***

Automatise les transformations des fichiers

```
#!/bin/sh

# Si le répertoire dont le nom est passé en paramètre, on supprime tous les
# fichiers contenus dans le dossier puis le dossier en lui-même
if [ -d $1 ] ; then
      rm -rf $1
fi

# Création du répertoire
mkdir $1
echo "Shell de mise en forme pour affichage"
echo "Tape sur <return>"

read junk
# Exécution du programme parse pour les trois fichiers récupérés
./parse tempo_1.txt chrono_1.dat
./parse tempo_2.txt chrono_2.dat
./parse tempo_3.txt chrono_3.dat

# Copie des fichiers dans le dossier créé pour faire de l'archivage
mv chrono_1.dat $1/chrono_1.dat
mv chrono_2.dat $1/chrono_2.dat
mv chrono_3.dat $1/chrono_3.dat
cp script.m $1/script.m

echo "J'ai fini !!"
```

Fichier *parse.c*

Transforme les fichiers de données en fichiers de données affichables en chronogrammes

```
#include <stdio.h>
#include "constantes.h"

FILE* temps,*chrono;

int get_debut();
int get_fin();

// On récupère dans la ligne de commande, le nom du fichier à lire
// et le nom du fichier où écrire
int main(int argc, char *argv[])
{
	long debut,fin;
	char buf[50];
	long int t, fin_count;
	int val;

	// On ne prend en compte que 15 périodes d'échantillonnage
	fin_count = 15 * (Te*1000);

	// Ouverture des fichiers
	temps = fopen(argv[1],"r");
	chrono = fopen(argv[2],"w");

	val = 0;
	// On initialise la variable début à la première valeur
	// de début contenue dans le fichier
	debut = get_debut();
	// On initialise la variable fin à la première valeur
	// de fin contenue dans le fichier
	fin = get_fin();
	t = 0;

	while(t<fin_count)
	{
		if(t==debut)
		{
		// Une fois arrivé à t=début on récupère la valeur suivante de début
		// On place un point de valeur 0 à une date t-1 avant la changement
		// On place un point de valeur 1 à la date t=début
			debut = get_debut();
			sprintf(buf,"%d\t%d\n",t-1,0);
			fwrite(&buf,sizeof(char),strlen(buf),chrono);
			sprintf(buf,"%d\t%d\n",t,1);
			fwrite(&buf,sizeof(char),strlen(buf),chrono);
		}
		if(t==fin)
		{
		// Une fois arrivé à t=fin, on récupère la valeur suivante de fin
		// On place un point de valeur 1 à une date t-1 avant la changement
		// On place un point de valeur 0 à la date t=fin
			fin = get_fin();
			sprintf(buf,"%d\t%d\n",t-1,1);
			fwrite(&buf,sizeof(char),strlen(buf),chrono);
			sprintf(buf,"%d\t%d\n",t,0);
			fwrite(&buf,sizeof(char),strlen(buf),chrono);
```

```
            }

            t = t+1;
      }
      // Fermeture des fichiers
      fclose(temps);
      fclose(chrono);
}



// Procédures de lecture de la valeur de début
int get_debut() {
      char tmp[50],nb[50];
      int i =0;

      fread(tmp,1,sizeof(char),temps);

      // Récupère le nombre de debut
      if(tmp[0] != 'd') return;
      i=0;

      while(tmp[0] != '\n')
      {
            fread(tmp,1,sizeof(char),temps);
            if(tmp[0]!='\n')
            {
                  nb[i] = tmp[0];
                  i++;
            }
      }
      nb[i] = '\0';
      return atoi(nb);
}

// Procédures de lecture de la valeur de fin
int get_fin() {
      char tmp[50],nb[50];
      int i =0;

      fread(tmp,1,sizeof(char),temps);
      // Récupère le nombre de fin
      if(tmp[0] != 'f') return;
      i=0;
      while(tmp[0] != '\n')
      {
            fread(tmp,1,sizeof(char),temps);
            if(tmp[0]!='\n')
            {
                  nb[i] = tmp[0];
                  i++;
            }
      }
      nb[i] = '\0';
      return atoi(nb);
}
```

Script Matlab *script.m*

Ce script permet de tracer les chronogrammes qui représentent les ordonnancements des trois tâches RTAI.

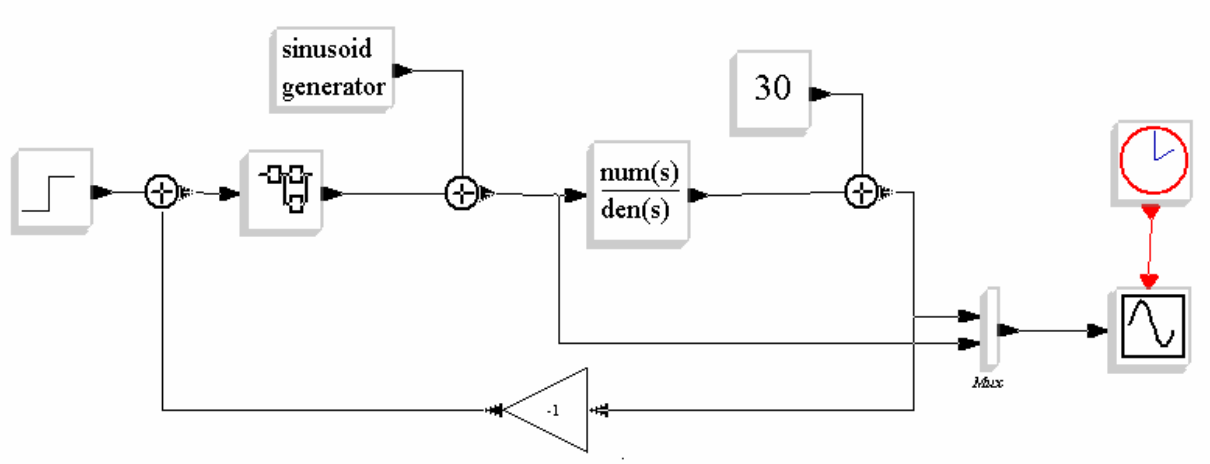
```
clear all
close all

% Périodes d'échantillonage
Te=1e3;
Periode_1=Te;
Periode_2=0.8*Te;
Periode_3=0.5*Te;

max = 9e3;

% Ouverture des fichiers en lecture
filename='chrono_1.dat';
fid1 = fopen(filename, 'rt')
filename='chrono_2.dat';
fid2 = fopen(filename, 'rt')
filename='chrono_3.dat';
fid3 = fopen(filename, 'rt')



% On lit chaque fichier pour en extraire les coordonnées de chaque point
i=1;
while feof(fid1) == 0
   tline1 = fgetl(fid1);
   [a(i),b(i)]=strread(tline1,'%d%d');
   i=i+1;
end

i=1;
while feof(fid2) == 0
   tline2 = fgetl(fid2);
   [c(i),d(i)]=strread(tline2,'%d%d');

   i=i+1;
end

i=1;
while feof(fid3) == 0
   tline3 = fgetl(fid3);
   [e(i),f(i)]=strread(tline3,'%d%d');
   i=i+1;
end

% On cherche à tracer les demandes d'activation pour chaque tâche
i=0;
t=1;
while(i*Periode_1 < max)
    p1x(t)=(i*Periode_1)-1;
    p1y(t)=0;
    p1x(t+1)=(i*Periode_1);
    p1y(t+1)=1;
    p1x(t+2)=(i*Periode_1)+1;
    p1y(t+2)=0;
    i=i+1;
```

```
    t=t+3;
end

i=0;
t=1;
while(i*Periode_2 < max)
    p2x(t)=(i*Periode_2)-1;
    p2y(t)=0;
    p2x(t+1)=(i*Periode_2);
    p2y(t+1)=1;
    p2x(t+2)=(i*Periode_2)+1;
    p2y(t+2)=0;
    i=i+1;
    t=t+3;
end

i=0;
t=1;
while(i*Periode_3 < max)
    p3x(t)=(i*Periode_3)-1;
    p3y(t)=0;
    p3x(t+1)=(i*Periode_3);
    p3y(t+1)=1;
    p3x(t+2)=(i*Periode_3)+1;
    p3y(t+2)=0;
    i=i+1;
    t=t+3;
end

% On créer un affichage multiple, pour voir tous les chronogrammes sur une
% seule figure
subplot(3,1,1);
hold on
plot(a,b,'b')
plot(p1x,p1y,'c')
axis([0 max 0 1.3])

subplot(3,1,2);
hold on
plot(c,d,'g')
plot(p2x,p2y,'c')
axis([0 max 0 1.3])

subplot(3,1,3);
hold on
plot(e,f,'k')
plot(p3x,p3y,'c')
axis([0 max 0 1.3])

% Fermeture des fichiers
fclose(fid1);
fclose(fid2);
fclose(fid3);
```

Expérimentation avec commande en Sinus

Schéma sous Scicos

La commande est consititué d'un sinus d'amplitude 20 rad/s et de valeur moyenne 120 rad/s.

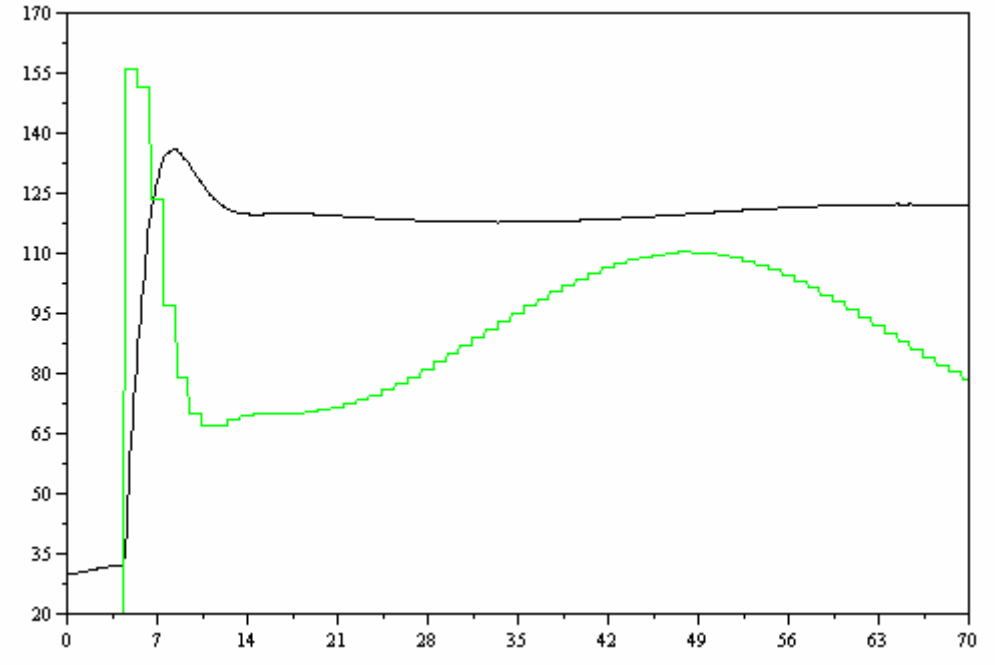


Résultats théoriques

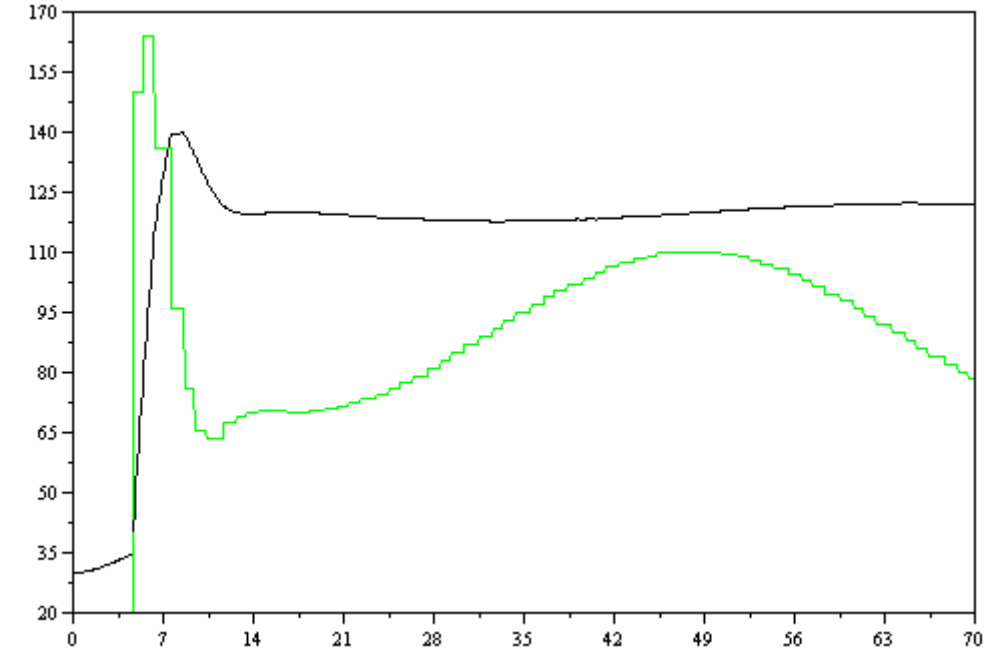
Jeu de tâches 1

- Non Préemptif

Objectif 120
Initiale 30



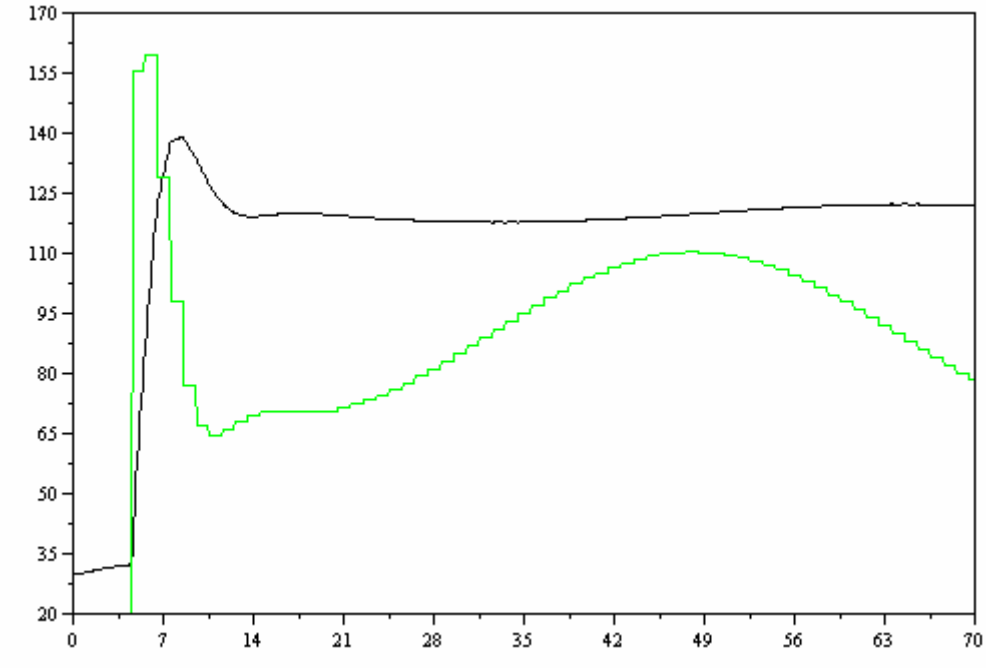
Gigue et retard Moyen



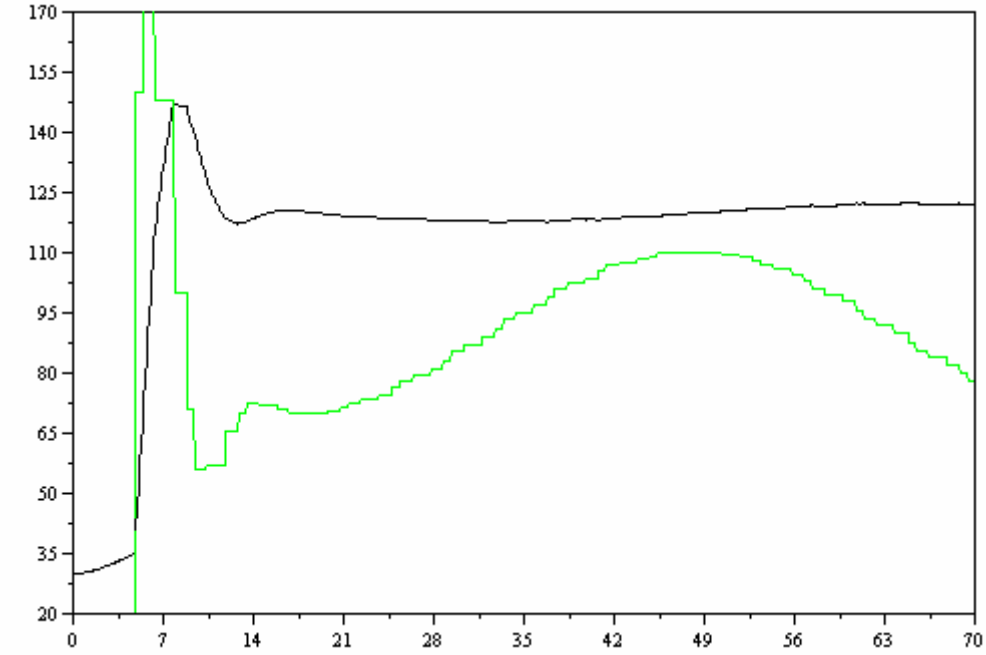
Gigue et retard Exact

- Préemptif

Objectif 120
Initiale 30



Gigue et retard Moyen

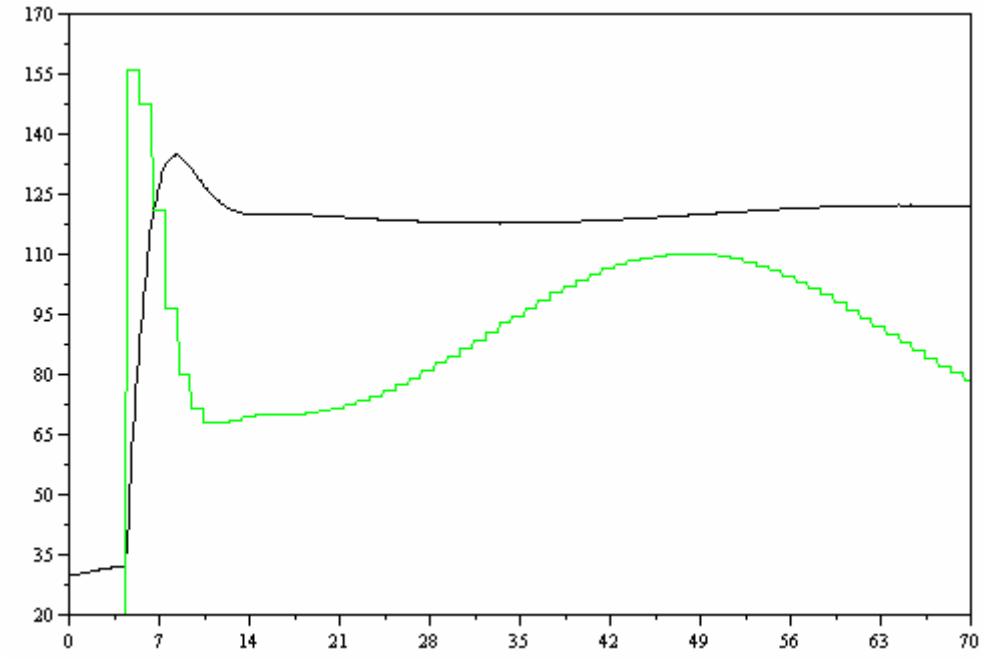


Gigue et retard Exact

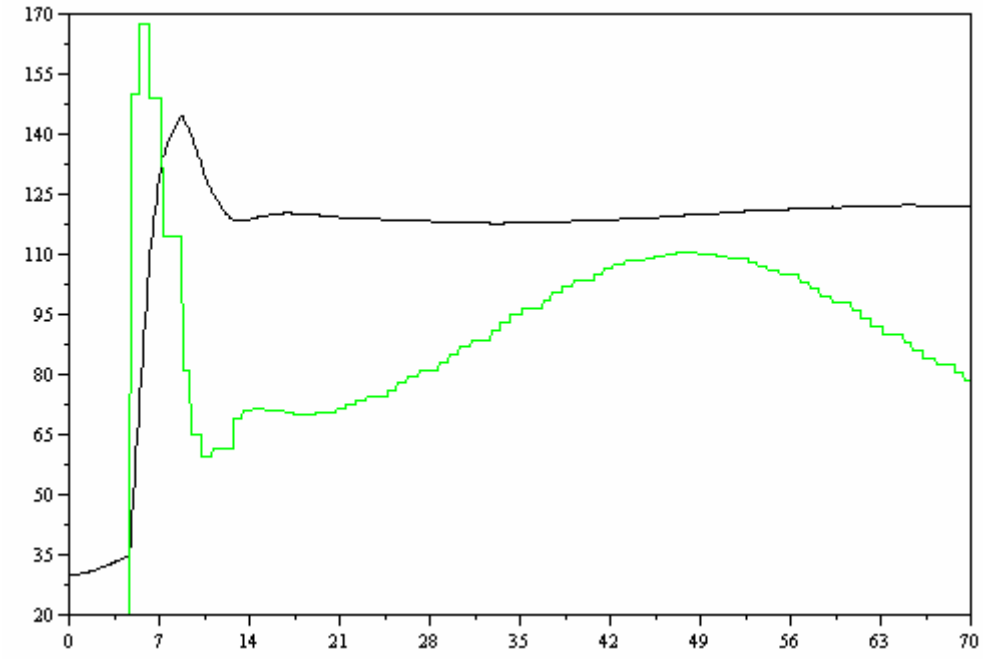
Jeu de tâches 2

- Non Préemptif et Préemptif

Objectif 120
Initiale 30



Gigue et retard Moyen : G =0.3 et R=0.1

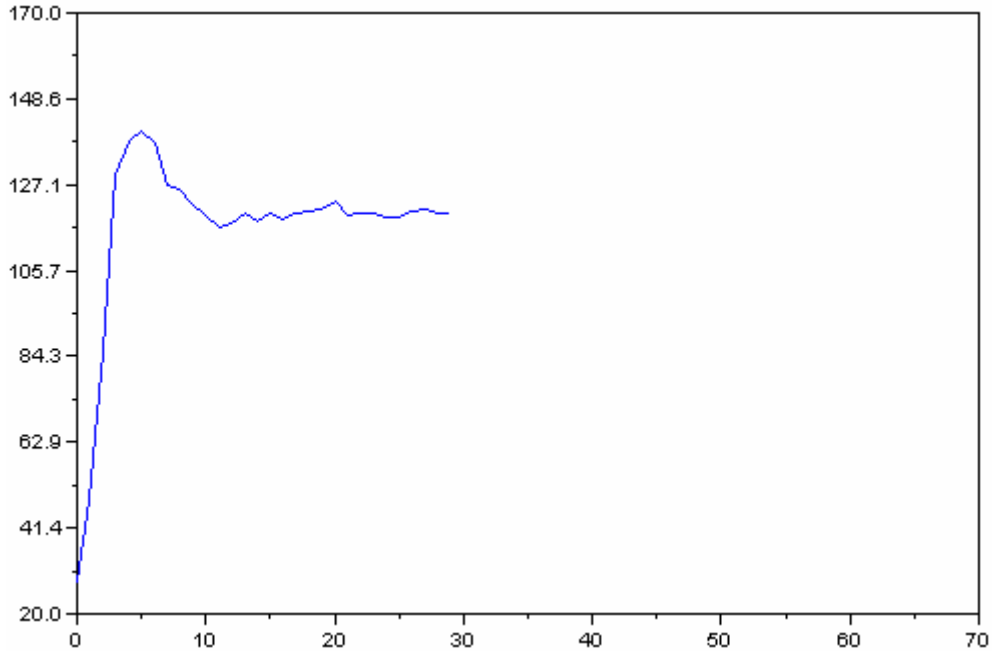


Gigue et retard Exact

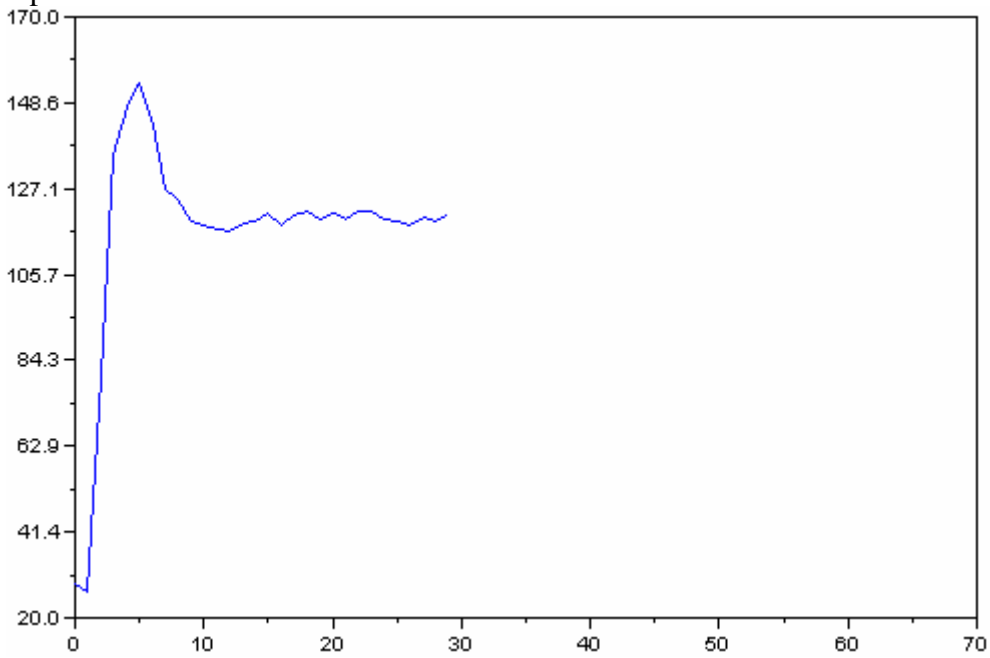
Résultats expérimentaux

Jeu de tâches 1

- Non Préemptif

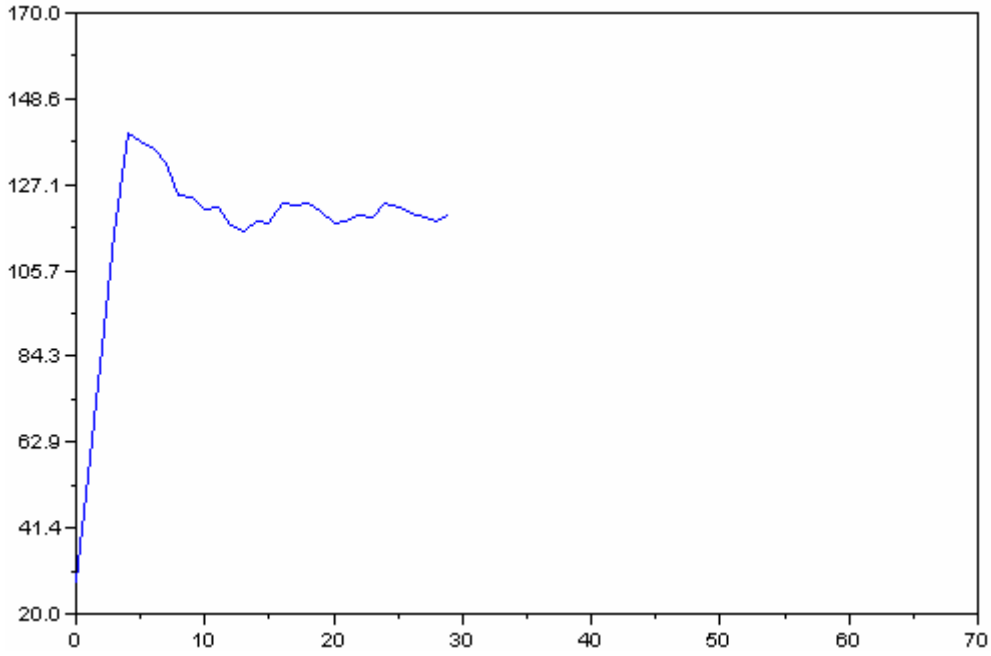
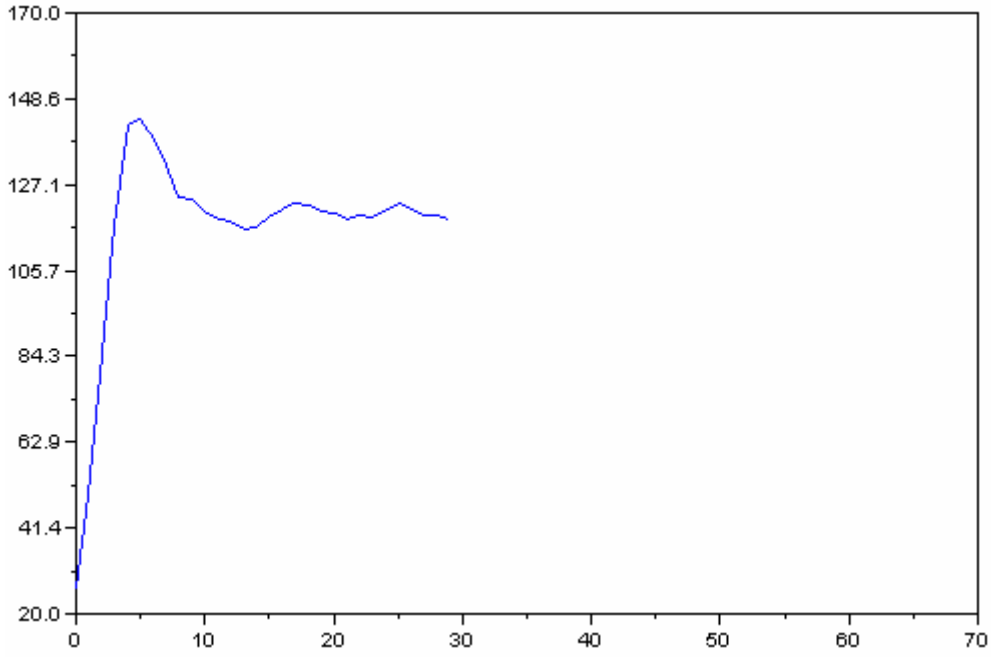


- Préemptif



Jeu de tâches 2

- Non Préemptif



- Préemptif