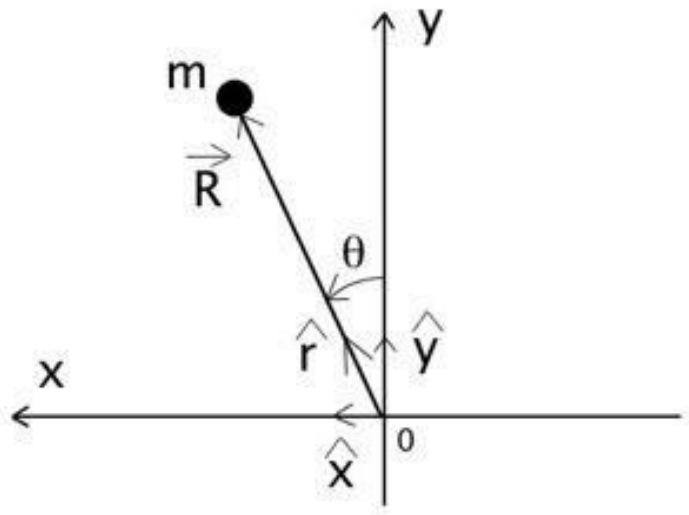
Rémi Bujon
Pierre Mollard

-Rapport-

Contrôle d'un pendule inversé



Rémi Bujon et Pierre Mollard

Ressources disponible sur GitHub
Septembre - Décembre 2020

Rémi Bujon
Pierre Mollard

Sommaire :

1. Présentation de l'étude de cas
2. Approche modulaire : notre architecture
3. Répartition du travail
4. Simulation
    a. Simulation sous matlab/simulink d'un pendule
    b. Simulation d'un pendule truetime
    c. Simulation pendule avec modules
5. Module Entrée/Sortie
    a. Développement du module spécifique de gestion entrée
    b. Développement du module spécifique de gestion sortie
    c. Test du module de gestion sortie
6. Module Acquisition/Restitution
    a. Développement de l'acquisition
    b. Développement de la restitution
    c. Test du module
7. Module Contrôleur
    a. Développement du module de contrôle
    b. Test du module de contrôle
    c. Test sur maquette
    d. Optimisations potentielles & problèmes
8. Module Communication
    a. Développement du module de contrôle bus CAN
    b. Test du bus CAN
9. Architecture fonctionnelle sur 2 pendules croisés
    a. Nouvel agencement module
    b. Test sur 2 pendules

Ressources disponible sur [GitHub]

Rémi Bujon
Pierre Mollard

1. Présentation de l’étude de cas :

L'objectif de cette étude est de parcourir l'ensemble d'un cycle de développement d'un système embarquable réparti et temps réel, de l'analyse à la réalisation d'un prototype opérationnel. Au terme de cette étude, notre système intégrera des procédés réels à commander, leurs interfaces électroniques, des procédés simulés à commander, plusieurs processeurs assurant différentes fonctionnalités (acquisition, conversion de commande, contrôleur ...), un réseau de communication entre des processeurs.

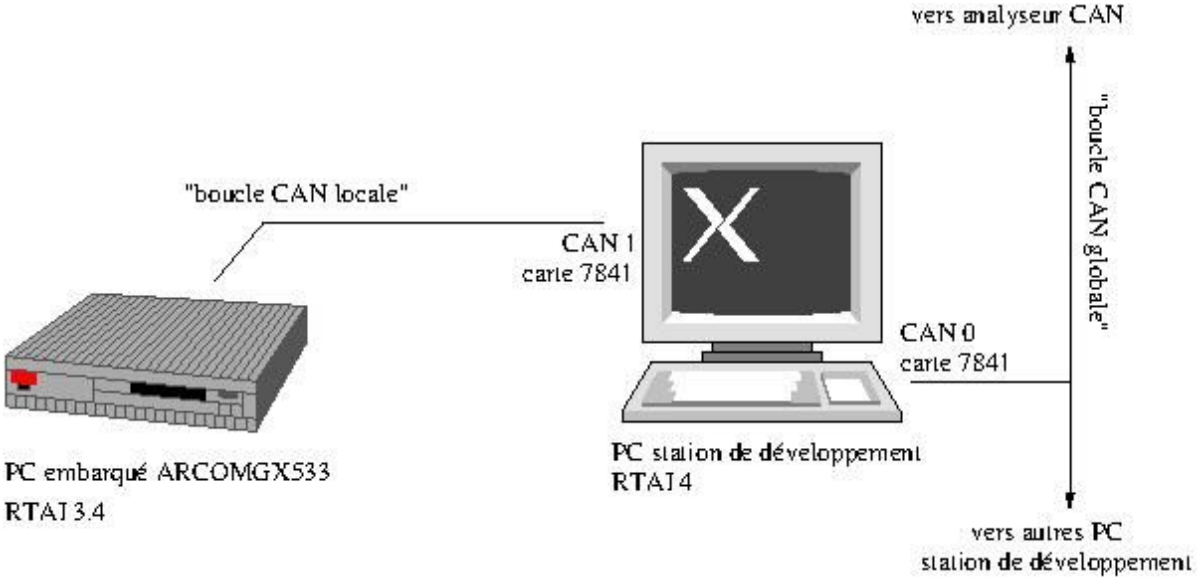
Durant l'étude de cas, nous nous attacherons à d'abord établir des modèles fonctionnels et à en simuler le comportement avant de passer à la réalisation. Tout élément devra être testé et validé.

Pour mener à bien notre étude de cas, nous avons à notre disposition :

- ❏ Une station de développement
- ❏ Un/plusieurs PC embarqués
- ❏ Une carte de communication CAN ADLINK7841 PCI
- ❏ Une carte de communication AIMPC104
- ❏ Une carte d’acquisition ADC
- ❏ Une carte d’acquisition DAC
- ❏ Un/Deux pendules inversés géographiquement distants

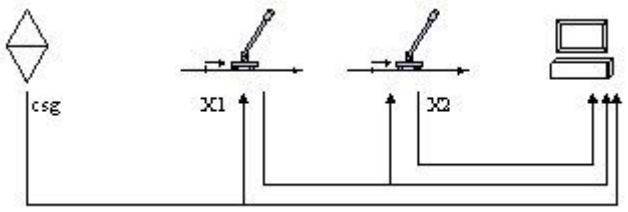
La station de développement de type PC sous Linux RTAI 4 communique avec un PC embarqué ARCOMGX533 sous Linux RTAI 3.4 via un bus CAN à l'aide d'une carte de communication CAN ADLINK 7841 PCI.

Cette même carte de communication nous permet d’avoir accès à un autre bus CAN indépendamment du précédent. Ce dernier aura une fonction plus globale et nous permettra de communiquer avec d’autres stations de développement.



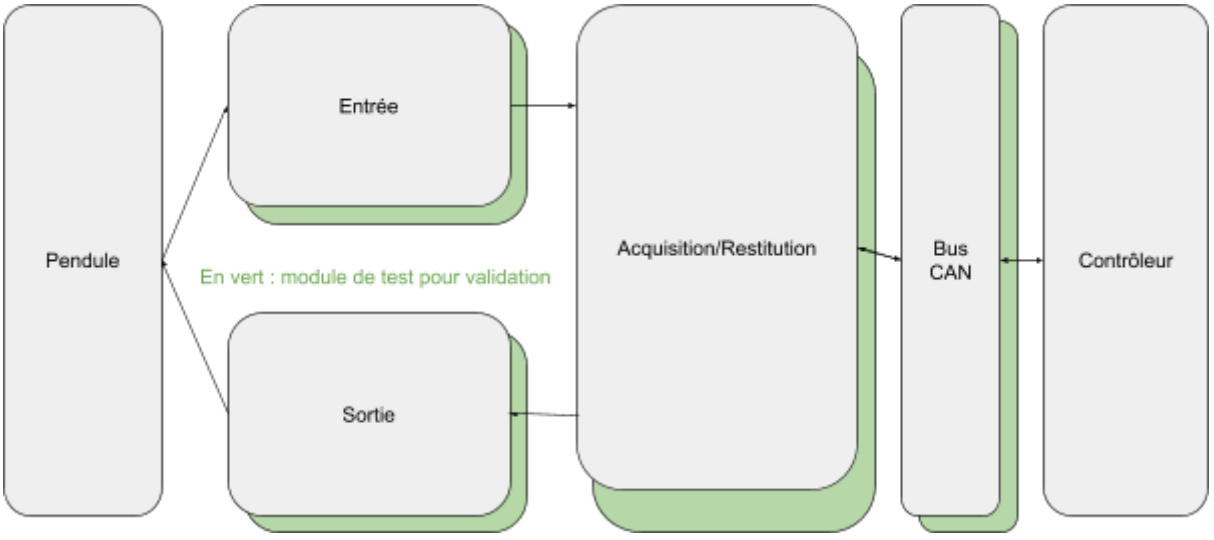
Les programmes seront développés et compilés sur la station de développement. Ils seront ensuite téléchargés sur le PC embarqué via une connexion ethernet.
Le PC embarqué intègre les 2 cartes d'acquisition ainsi que la carte communication CAN AIMPC104.

Chaque PC embarqué est ensuite connecté à un pendule inversé. Le pendule inversé nous renvoi deux valeurs, l'angle du pendule, et sa position sur le rail.
La seule valeur sur laquelle nous pouvons jouer pour équilibrer le pendule est celle sur sa position.



2. Approche modulaire :

Pour commander ce pendule nous choisissons une approche modulaire afin de corriger plus facilement les erreurs et de maîtriser complètement la chaîne d'acquisition.
De plus, l'approche modulaire va nous permettre d'intégrer un 2ème pendule plus facilement par la suite. Nous pouvons aussi définir quel calculateur effectue quelle tâche (1 tâche par module) grâce au module de communication associé au bus CAN.



Les modules Entrée et Sortie sont liés aux cartes ADC ADVANTECH 3718HG PC104 et DAC ADVANTECH PCM-3712 PC104 respectivement. Ces modules initialisent les cartes et possèdent chacun une tâche périodique unique afin de contrôler les signaux envoyés aux actionneurs et les signaux envoyés par les capteurs de manière asynchrone.

Ces modules sont génériques et compatibles avec n'importe quelle carte du même type que la 3718 et 3712, ils nécessitent les bibliothèques 3718.h et 3712.h embarqués sur les bonnes cartes.
Cette approche permet d'éviter de redéfinir les consignes d'acquisition lorsqu'on ajoute des pendules au système.

Le module Acquisition/Restitution définit le nombre de channel et la plage des informations transmises par le contrôleur ou reçu depuis les capteurs. Il permet aussi d'ajuster les fréquences d'échantillonnage/d'envoi des informations.
Ce module est spécifique au système, il faut faire attention au numéro des channels et aux valeurs des plages afin que les entrées et sorties entre la carte et le pendule correspondent. Par exemple, si le channel 0 de la carte est connecté électriquement au capteur de position, alors il faut définir dans le logiciel (module Entrée) que l'information de position est récupérée en channel 0.

Le module bus CAN permet de gérer la communication entre les différents composants du système. Il est lancé sur la carte de communication CAN AIMPC104.
Dans un premier temps nous développons un seul pendule sans besoin de communication externe. Ce module sera utilisé dans le but d'ajouter un 2ème pendule et de croiser les tâches et les calculateurs.

Le module Contrôleur va effectuer les calculs nécessaires afin de contrôler le pendule et le stabiliser, il va prendre en compte les informations en Entrée afin de générer une commande en Sortie. Ce module est exécuté sur un PC embarqué : une carte ARCOM SBCGX533.

3. Répartition du travail :

Pour réaliser ce travail, nous allons dans un premier temps réaliser une simulation du pendule sous simulink à l'aide d'un modèle physique du système que nous allons contrôler par approche Observateur/Contrôleur.

Nous commencerons par une phase 1 pour contrôler un unique pendule sans communication dans différents calculateurs.
Puis nous simulons des processeurs pour mettre en place une communication entre observateur et contrôleur à l'aide de truetime pour simulink.
Enfin nous simulons la pendule avec des contrôleurs et observateurs sur deux calculateurs différents afin de montrer la rigueur du protocole de communication.

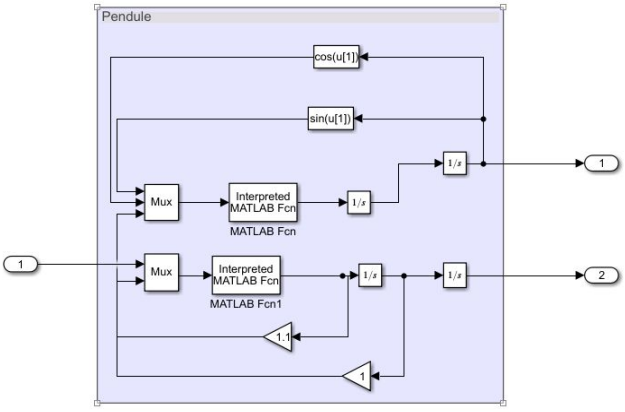
Après simulation, nous allons développer les modules d'acquisition et restitution avec leur module de test respectif afin de s'assurer d'avoir des signaux valide dans notre système.

Nous pourrons ensuite développer le module Entrée avec son module de test respectif afin de s'assurer d'avoir des informations valident issues de nos capteurs.
De même avec le module Sortie, nous le testerons sur oscillateurs pour vérifier que l'information électrique est cohérente avec les besoins du moteur.

Finalement le module Contrôleur sera le même code que celui développé sous matlab, traduit en C avec une gestion de la mémoire précise et une prise en compte du temps réel de calcul. Certains paramètres pourront être ajustés comme le gain en sortie.

4. Simulation :

Pour simuler le contrôle du pendule, on s'appuie sur le modèle physique ci-dessous :



L'objectif va être de mettre en place le modèle du contrôleur basé sur l'observateur.

a. Simulation sous matlab/simulink d'un pendule

Il faut commencer par linéariser le modèle du pendule aux conditions initiales afin d'obtenir les matrices d'état A, la matrice de commande B, la matric d'observation C et la matrice d'action directe D (nulle pour nous).

```
x_0=[0 0 0 0]';
[A, B, C, D]= linmod('ModelePendule',x_0);
```

On choisit les pôles du contrôleur qui sont les valeurs propres de la matrice d'état.
On choisit des pôles 3 fois plus rapide pour l'observateur pour s'assurer que les valeurs observées soit les plus récentes possible pour le contrôleur.

```
Pc=[-3,-8,-2.5,-2];
Po=3*Pc;
```

Le contrôle se fait en discret, il faut donc échantillonner les matrices avec un pas d'échantillonnage choisi à 0.01s. On utilise la fonction matlab c2d pour y arriver.

```
T_ech=0.01;
[Ad,Bd]=c2d(A,B,T_ech);
Cd=C;
Dd=D;
```

Rémi Bujon
Pierre Mollard

De même, l'échantillon implique une transformation des pôles :

$$Poles_d = exp^{(Poles \times T_{ech})}$$

Soit :

```
Poles_d=exp(Pc*T_ech);
Poles_od=exp(Po*T_ech);
```

On utilise ensuite la fonction matlab place() afin de placer les pôles en fonction de notre système et de trouver le gain de correction nécessaire en boucle fermée K.

```
Kcd=place(Ad,Bd,Poles_d);
Kod=place(Ad',Cd',Poles_od)';
```

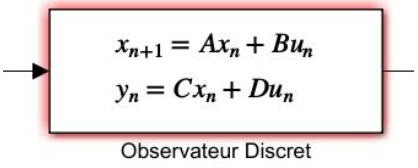
On calcul les matrices vues par l'observateur en utilisant le modèles vu en TP e4 :

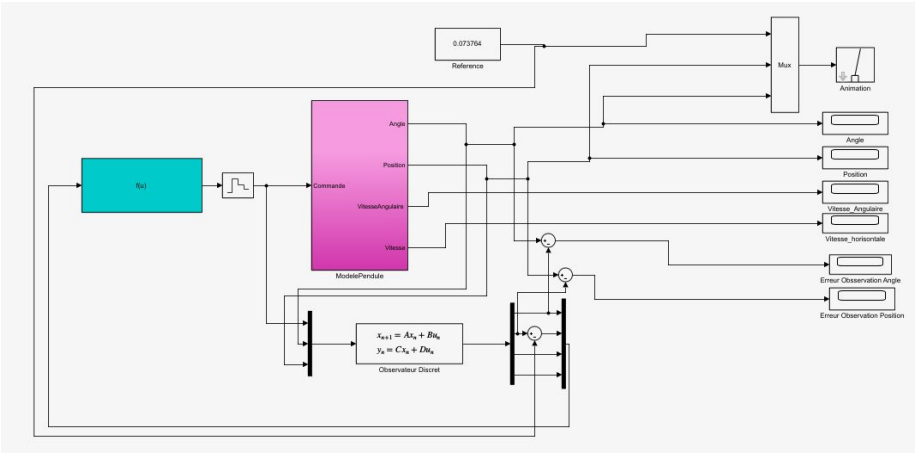
$$\begin{cases} \hat{x}(k+1) = (A_d - K_{od}C_d)\hat{x}(k) + [B_d, K_{od}][u(k), y(k)]^T \\ \hat{y}(k) = C_d\hat{x}(k) \end{cases}$$

Nous pouvons ainsi sortir :

```
Aobs_d=Ad-Kod*Cd;
Bobs_d=[Bd Kod];
Cobs_d=eye(4,4);
Dobs_d=zeros(4,3);
```

On lance le script matlab afin d'obtenir ces 4 matrices, puis on utilise le bloc simulink Observateur Discret afin de calculer itérativement les valeurs du vecteur d'état et de commande :



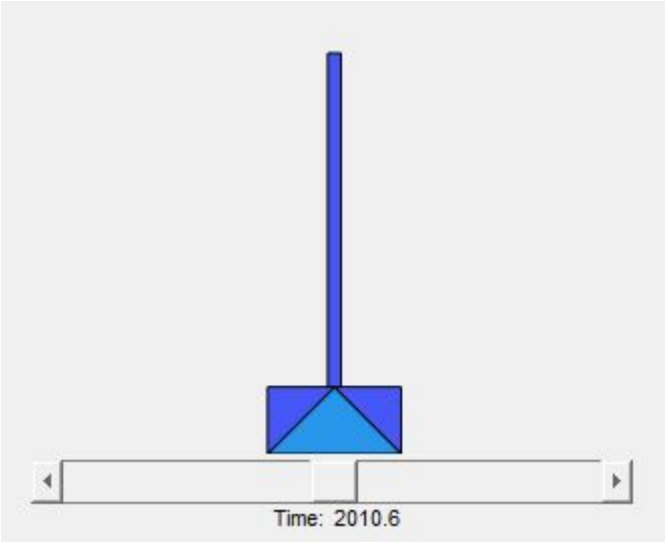


Le modèle ci-dessus isole la commande, la discrétise pour simulink et l'envoie au modèle.

Rémi Bujon
Pierre Mollard

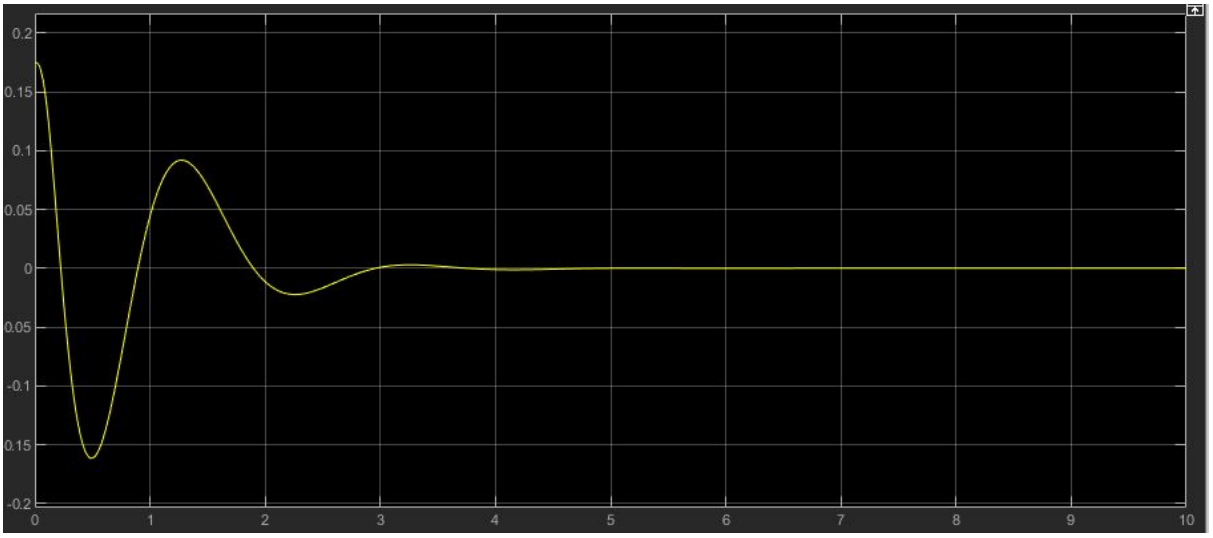
Il ne nous reste plus qu'à lancer le script, d'observer et d’analyser les résultats obtenus. A la base le pendule n’est pas à sa position géographique souhaité sur son axe. Nous estimons donc qu’il va se déplacer vers cette nouvelle valeur.

La première chose que nous pouvons observer est une simulation en 2D du pendule, de ses mouvements et de sa position angulaire.



Simulation 2D du pendule

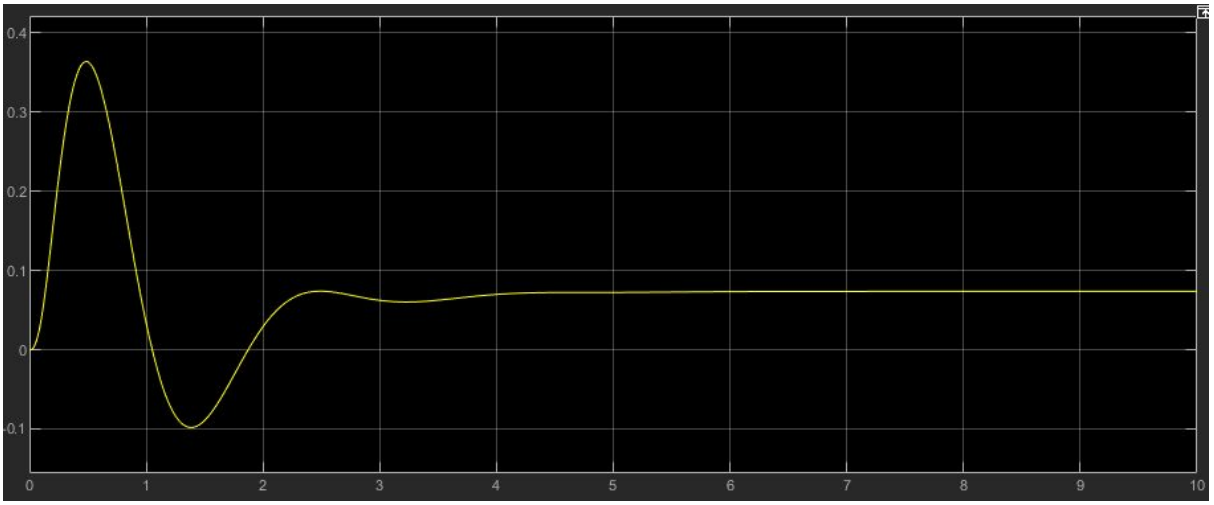
Nous pouvons observer que le pendule s’équilibre bien à la verticale au bout de 5s et qu’il ne bouge absolument plus une fois qu’il atteint sa position finale, en tout cas à vue d'œil. Cependant, nous allons vérifier ces observations à l’aide de graphiques plus parlants.



Position angulaire du pendule

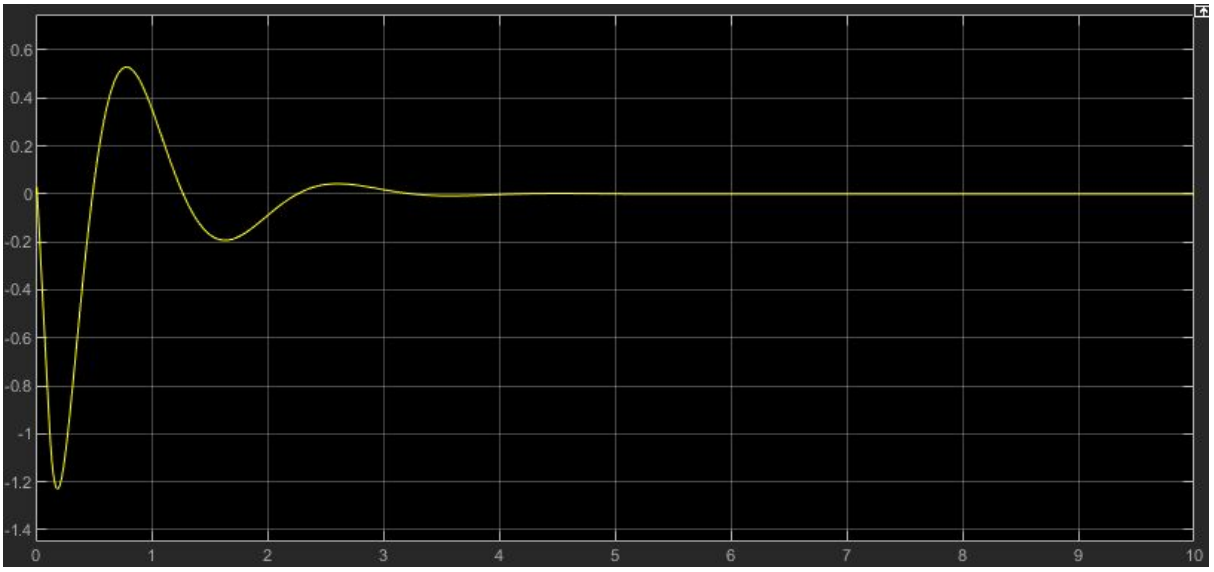
Rémi Bujon
Pierre Mollard

Nous représentons ici la position angulaire du pendule en fonction du temps. Au démarrage du programme, nous pouvons observer de fortes variations de l'angle. C'est tout à fait normal car le pendule cherche à s'équilibrer à sa position stable à la verticale. Une fois sa position finale atteint à une valeur de 0 au bout d'environ 5s, ce dernier ne varie plus et donc a atteint sa position finale.



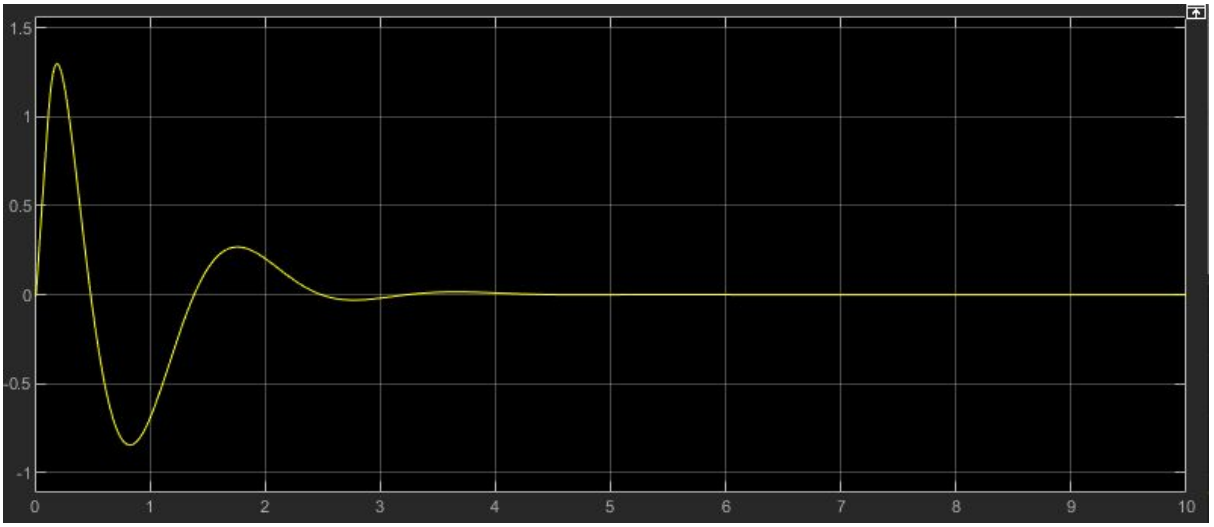
Position linéaire du pendule

Nous représentons ici la position du pendule sur son axe. Au début, ce dernier se trouve au centre de son axe. Cependant, nous avons rentré une valeur de position souhaitée différente de cette dernière. Lorsque nous démarrons le scripte, nous pouvons à nouveau observer des variations sur l'axe du pendule, ce qui représente les mouvements de ce dernier pour s'équilibrer. Au bout d'environ 5s, ce dernier atteint sa position finale à x = 0,07 (positions du curseur, que nous pouvons faire varier à notre guise).



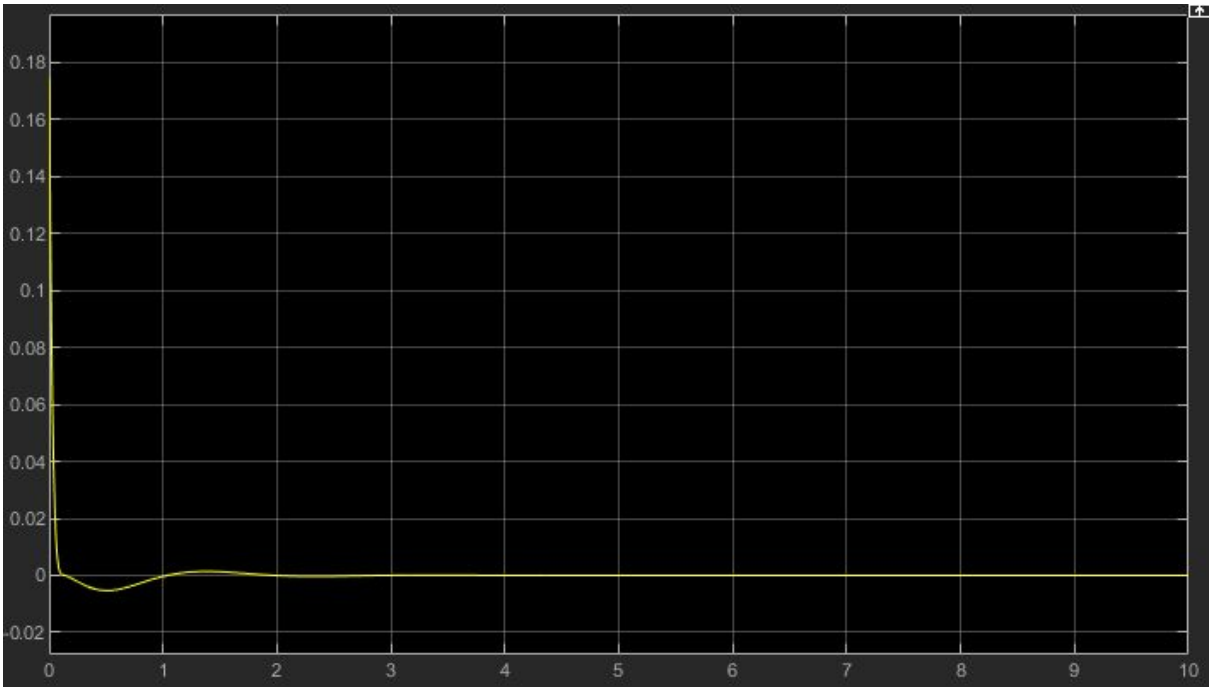
Vitesse angulaire :

Nous représentons ici la vitesse angulaire du pendule. Comme dans les observations précédentes nous pouvons à nouveau observer de fortes variations au début de l'acquisition puis une stabilisation à environ 5s. Le pendule est donc à l'arrêt et stabilisé une fois cette période passée.



Vitesse horizontale :

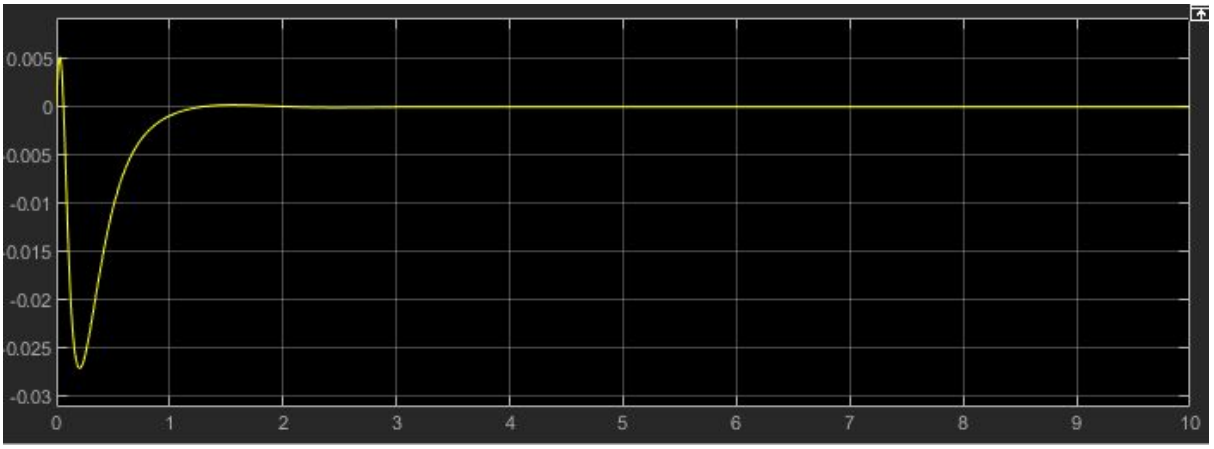
Nous représentons ici la vitesse horizontale du pendule. De même que les observations précédentes, nous pouvons observer les mêmes résultats. Ce coup-ci nous observons une forte accélération puis une décélération pour permettre au pendule encore une fois de se stabiliser.



Erreur observation angle :

Nous représentons ici l'erreur angulaire du pendule par rapport à sa valeur souhaitée en fonction du temps. Nous pouvons observer que au démarrage de l'acquisition, le pendule

n'est pas à la verticale mais plutôt avec un angle de 0.18 radian (environ 10 degré). Puis suite à l'exécution du programme, nous pouvons observer une correction très rapide de cette erreur en environ 0.1s. Puis il faudra environ 3s au système pour que l'erreur d'angle ne varie plus ou très peu. Cela nous fait comprendre que le pendule va d'abord corriger l'erreur de l'angle puis se déplacer doucement vers sa position horizontale souhaitée.



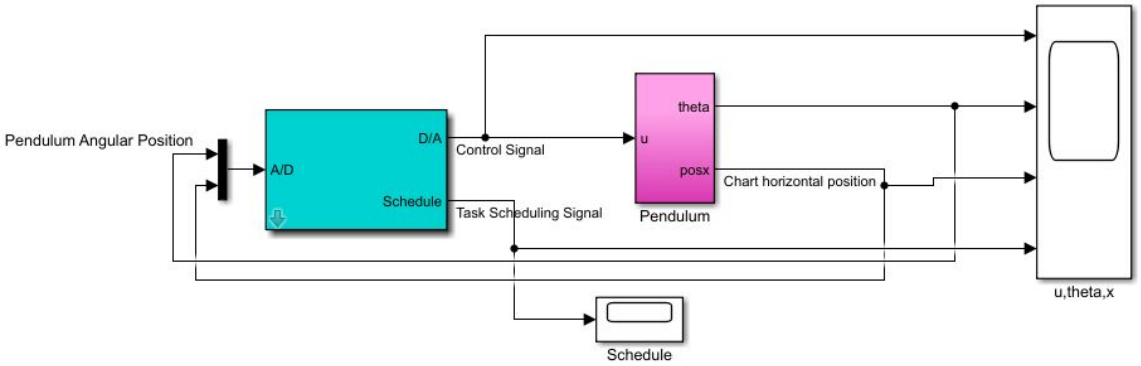
Erreur observation position :

Nous représentons ici l'erreur en position du pendule par rapport à sa valeur souhaitée en fonction du temps. Ce coup-ci, nous pouvons observer que le pendule va commencer par dépasser sa valeur finale très rapidement, puis se stabilise beaucoup plus doucement vers sa valeur finale. Cela rentre en corrélation avec l'observation précédente. Le pendule commence par corriger son angle et s'équilibrer pour ensuite se déplacer vers sa valeur en position souhaitée.

b. Simulation d'un pendule avec truetime

Pour pouvoir appliquer une approche modulaire, il nous faut installer et utiliser Truetime. Ce dernier nous permet de simuler en temps réel un système contrôlé par des processus simulés. Cela inclut un fonctionnement par blocs/modules.

On remplace donc les blocs de calcul du vecteur d'état et de commande par un processeur simulé :



On affiche les graphes de l'évolution des variables du pendule, ainsi que la charge du processeur.

A l'initialisation, le processeur truetime va lancer exécuter le code suivant :

On définit l'ordonnancement du processeur comme priorité fixé vu qu'il n'y a qu'une tâche

```
ttInitKernel('prioFP');
```

On charge en mémoire les matrices du système trouvé dans la partie précédente :

```
Adc=[0.6300   -0.1206   -0.0008    0.0086
     -0.0953    0.6935    0.0107    0.0012
     -0.2896   -1.9184    1.1306    0.2351
    -3.9680   -1.7733   -0.1546    0.7222];

Bdc=[0.3658    0.1200
    0.0993    0.3070
    1.0887    2.0141
    3.1377    1.6599];

Cdc=[-80.3092   -9.6237  -14.1215  -23.6260];

Ddc=[0 0];
```

On définit le temps d'échantillonnage Tech = 0.01s et la tâche de calcul comme étant périodique avec sa date limite égale à sa période (c'est une tâche unique).
Sans décalage car elle commence dès le début de la simulation, et sa priorité n'as pas d'importance.

```
Tech=0.01;
period = Tech;
deadline = period;
offset = 0.0;
prio = 1;
```

On charge les paramètres en mémoire local pour que le code de la tâche y ait accès en définissant des valeurs initiale pour les vecteur d'état, d'entrée/sortie et de commande :

```
data.Adc=Adc;
data.Bdc=Bdc;
data.Cdc=-Cdc;
data.Ddc=Ddc;

data.x=[0 0 0 0]';
data.y=[0 0]';
data.u=0;
```

On précise ensuite les valeurs initiales des channels d'entrées/sorties et le numéro de ces channels pour qu'il correspondent avec les connexions du modèle:

```
data.th = 0;
data.d=0;
data.thChanel=1;
data.dChanel=2;
data.uChanel=1;
```

On crée ensuite la tâche périodique avec la fonction de truetime en précisant le nom du fichier contenant son code d'exécution :

```
ttCreatePeriodicTask('Obs_Cont_task', offset, period, 'Obs_Cont', data);
```

Le code exécuté périodiquement par la tâche est le suivant :

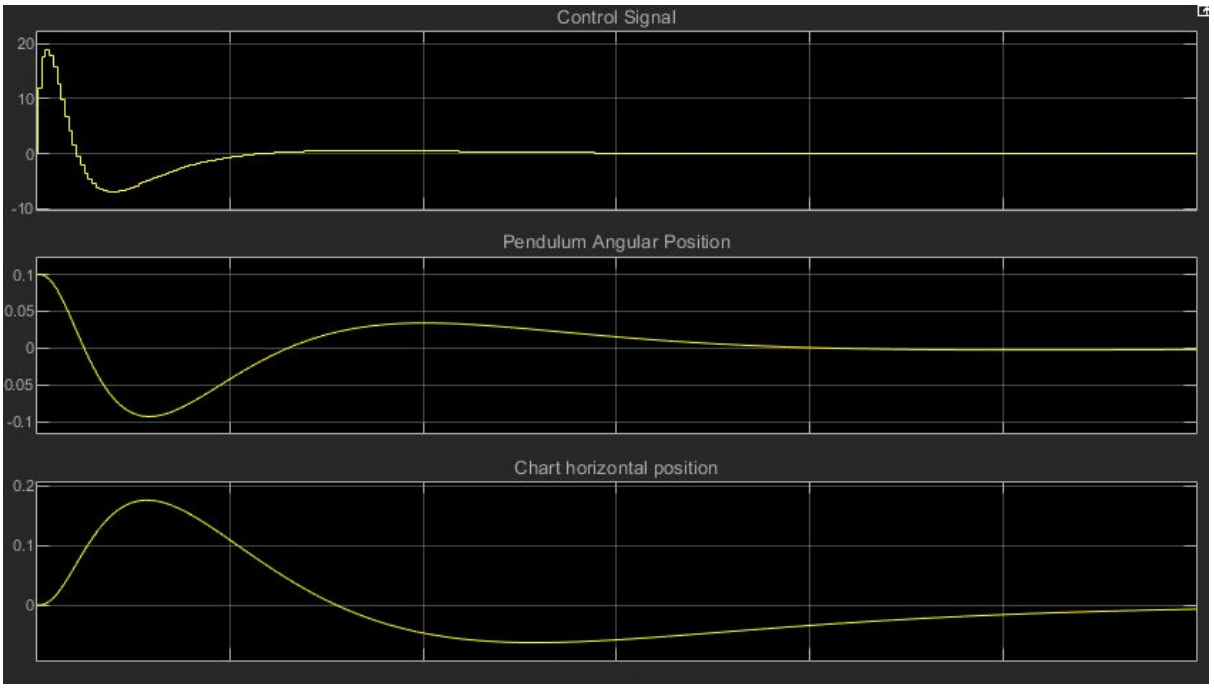
```
function [exectime, data] = Obs_Cont(seg, data)

switch seg

 case 1
  theta = ttAnalogIn(data.thChanel);
  d = ttAnalogIn(data.dChanel);
  y=[theta;d];
  data = obscont(data, y);
  exectime = 0.002;
 case 2
  ttAnalogOut(data.uChanel, data.u);
  exectime = -1;
end
```

Le code est divisé en 2 parties : gestion entrée puis gestion sortie.
Dans le 1er cas, la lecture des channels d'entrée est stockée dans les variables theta et d qui sont combinées dans le vecteur d'entrée y.
Le calcul de la commande est réalisé dans la méthode obscont(data, y) expliquée après.
Dans le 2ème cas, on écrit dans le channel de sortie la valeur de la commande stockée précédemment dans data.
Exectime permet d'estimer la durée de chaque étape, 0.002 représente 20% de la durée totale que peut prendre la tâche (définie arbitrairement), -1 signifie que la tâche est terminée.

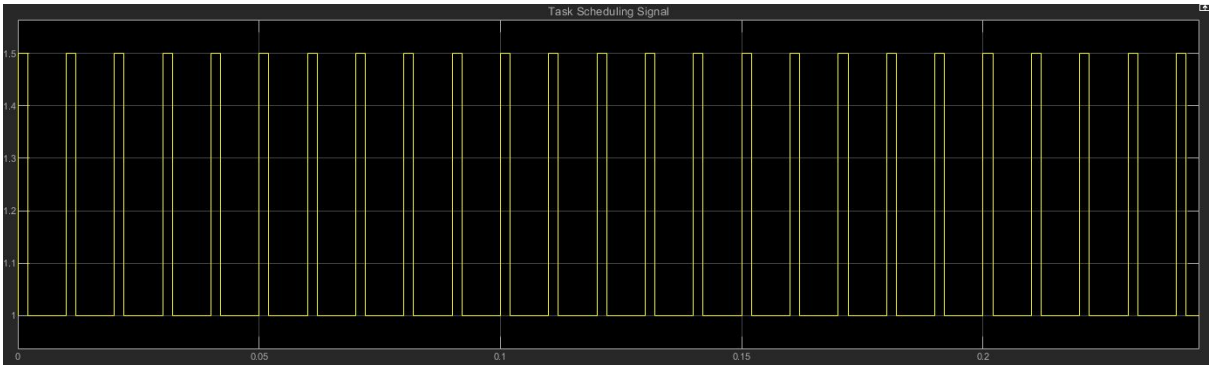
Le code de obscont reprend les équations du bloc précédent permettant de calculer le vecteur d'état et de commande et de stocker les résultats dans data :

```
function data = obscont(data, y)

data.x=data.Adc*data.x+data.Bdc*y;
data.u=data.Cdc*data.x;
```

Après exécution, on obtient les résultats suivants.



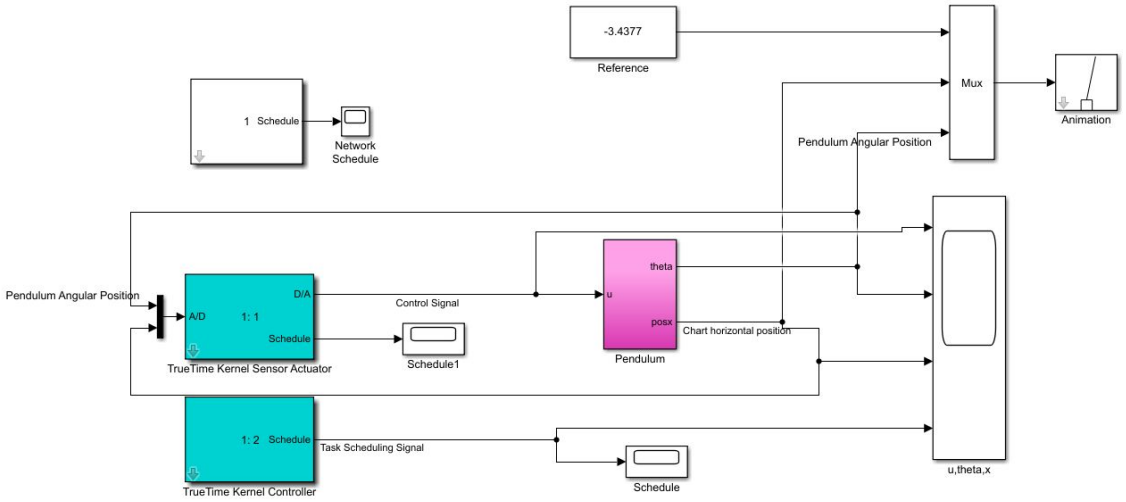
Nous pouvons observer 3 différents graphiques dans cette première capture. Tout d'abord, le premier graphique représente le signal de contrôle en fonction du temps. Nous pouvons remarquer que ce dernier est échantillonné avec une période de 0,01s. Nous n’observons plus de variations à partir d’environ 3s.
Ensuite, le second graphique représente la position angulaire du pendule en fonction du temps. Nous pouvons observer les mêmes résultats que lors du test précédent sans l’approche modulaire, c’est-à-dire avec de fortes variations au démarrage du script, puis une valeur finale à 0 à environ 5s.
Enfin, le dernier graphique représente la position horizontale du chariot par rapport au temps. Encore une fois, nous pouvons observer les mêmes résultats que lors du test précédant sans l’approche modulaire. Nous pouvons observer ce coup-ci une position finale au centre de l’axe horizontal, cependant cela prend légèrement plus de temps avec une position finale atteint après 6s.



Ce dernier graphique représente la planification des tâches du signal en fonction du temps. Nous pouvons observer 10 tâches en 1s avec une occupation du temps de 20% entre chaque début de tâche. C'est ce qui était attendu et souhaité lors de l'écriture du code.

c. Simulation pendules avec modules

Afin de respecter nos modules, nous allons séparer l'acquisition et restitution du calcul des vecteurs.



Nous avons 2 processeurs : un pour le calcul et l'autre pour la restitution/acquisition comme dans notre architecture de maquette.
Le premier va lancer le code d'initialisation controller_init et le deuxième actuator_init :

Controller_init est le même que précédemment, on a supprimé les parties concernant l'autre processeur et le code périodique à changer de reférence :

```
ttCreateTask('controller_task', deadline, 'Obs_Cont',data);
ttAttachNetworkHandler('controller_task')
```

On synchronise les 2 processeurs pour qu'il utilisent la même fréquence d'horloge en déclarant un réseau avec ttAttachNetworkHandler. Ce processeur est positionné sur le node 2.

On définit un ordonnancement DM (deadline monotonic), soit plus petite deadline en premier, en effet il faut que la tâche de contrôle se fasse après la tâche d'acquisition puis enfin la restitution.

```
ttInitKernel('prioDM')
```

Le processeur de calcul possède qu'une tâche :

```
function [exectime, data] = Obs_Cont(seg, data)

switch seg,
 case 1,
  data.y = ttGetMsg;
  data.x=data.Adc*data.x+data.Bdc*data.y;
  data.u=data.Cdc*data.x;
  exectime = 0.001;
 case 2,
  ttSendMsg(1, data.u, length(data.u));
  exectime = -1; % finished
end
```

Il s'agit d'une version simplifiée de la première itération. On ne récupère plus 'y' avec les deux channel d'entrée, il est récupéré par un message de la tâche précédente (acquisition). Le calcul obscont() est fait en local, on estime sa durée d'exécution à 10% de la période. Une fois le calcul fait, on envoie le vecteur commande calculé à la tâche suivante déclarer sur le node 1 (restitution). On précise la taille du message

Sur le node 0 et 1, on déclare 2 tâches : acquisition et restitution (sensor & actuator).
On garde une priorité d'ordonnancement DM.
On garde les channel précédent :

```
data.th = 0;
data.d=0;
data.y=[0;0];
data.u=0;
data.thChanel=1;
data.dChanel=2;
period = 0.010;
starttime = 0.0;
```

Puis on déclare nos 2 tâches (avec une deadline grande pour l'actuator car plus faible priorité):

```
ttCreatePeriodicTask('sensor_task', starttime, period, 'sensor_code',data);

% Sporadic actuator task, activated by arriving network message
deadline = 10.0;
ttCreateTask('actuator_task', deadline, 'actuator_code',data);
ttAttachNetworkHandler('actuator_task')
```

La restitution s'active lorsqu'il reçoit un message du contrôleur afin qu'il s'exécute toujours après.

La tâche d'acquisition :

```
function [exectime, data] = sensor_code(seg, data)

switch seg,
 case 1,
  th = ttAnalogIn(data.thChanel);
  d  = ttAnalogIn(data.dChanel);
  data.y=[th; d];
  exectime = 0.005;
 case 2,
  ttSendMsg(2,data.y,length(data.y)); % Send message (80 bits) to node 3 (controller)
  exectime = 0.0004;
 case 3,
  exectime = -1; % finished
end
```

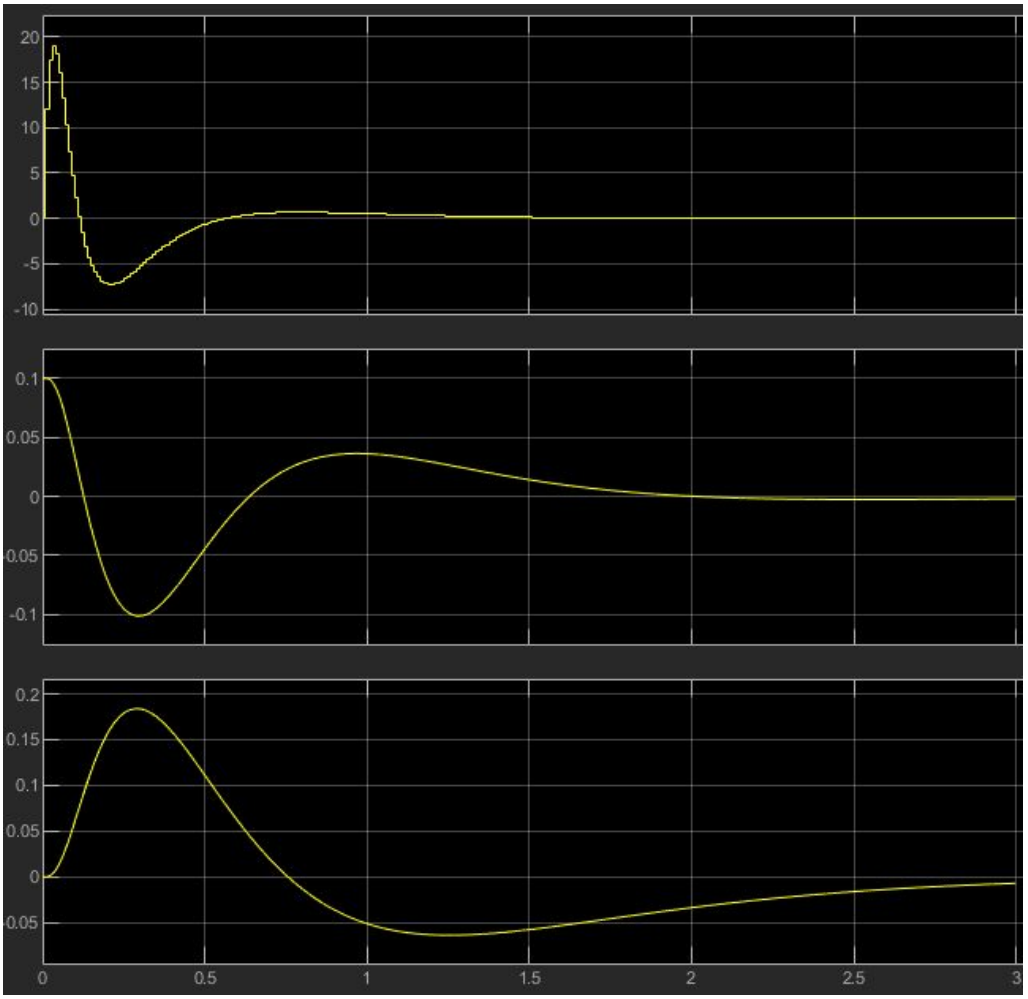
On récupère les informations des channels comme dans le cas monoprocesseur, au lieu de les traiter directement, on les envoie au contrôleur avec ttSendMsg.

La tâche de restitution :

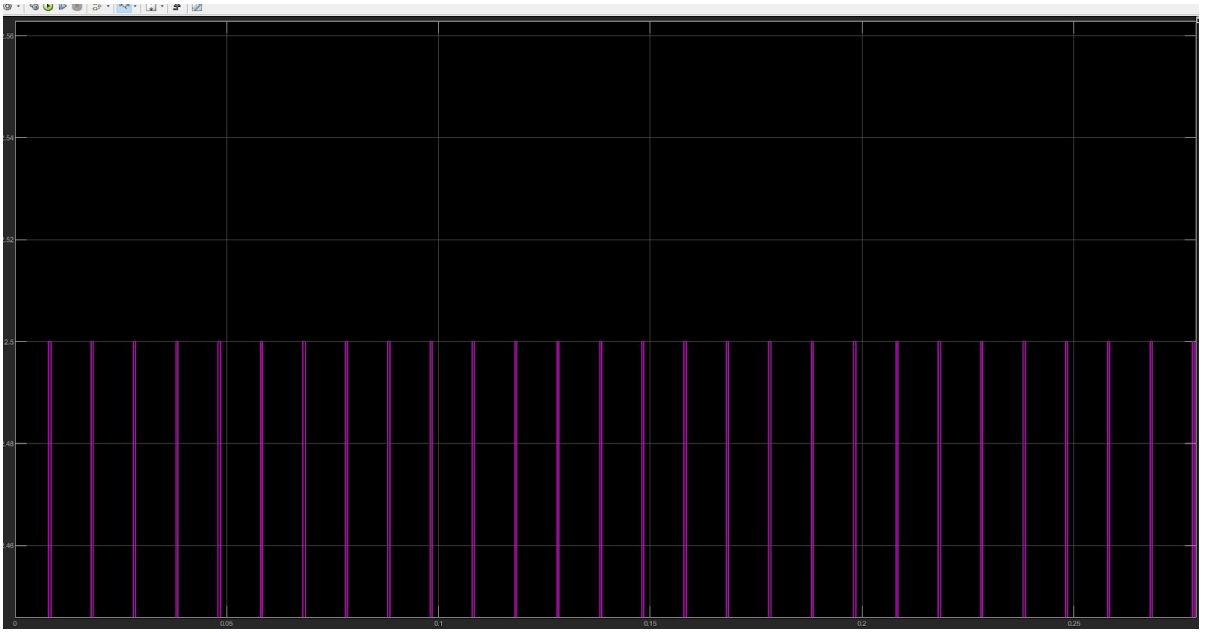
```
function [exectime, data] = actuator_code(seg, data)

switch seg,
 case 1,
  data.u = ttGetMsg;
  exectime = 0.0005;
 case 2,
  ttAnalogOut(1, data.u)
  exectime = -1; % finished
end
```

Elle s'active dès réception d'un message et envoie le contenu du message sur le channel correspondant à la sortie comme pour la fin de la tâche monoprocesseur.

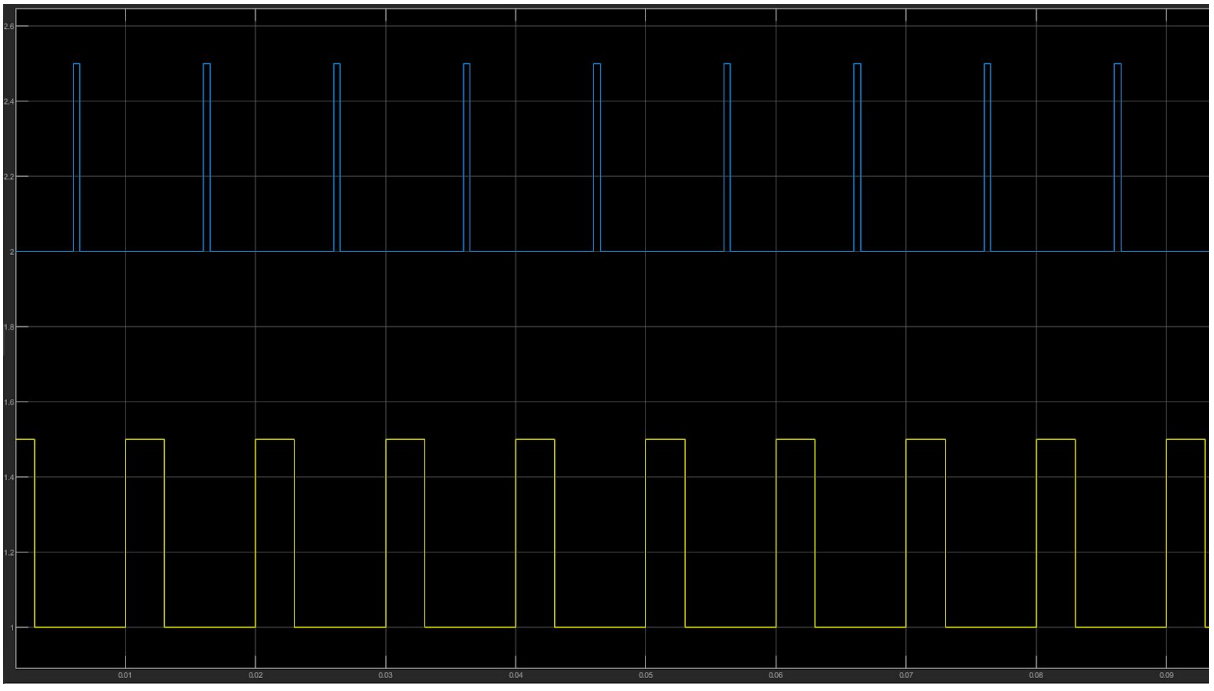
Rémi Bujon
Pierre Mollard



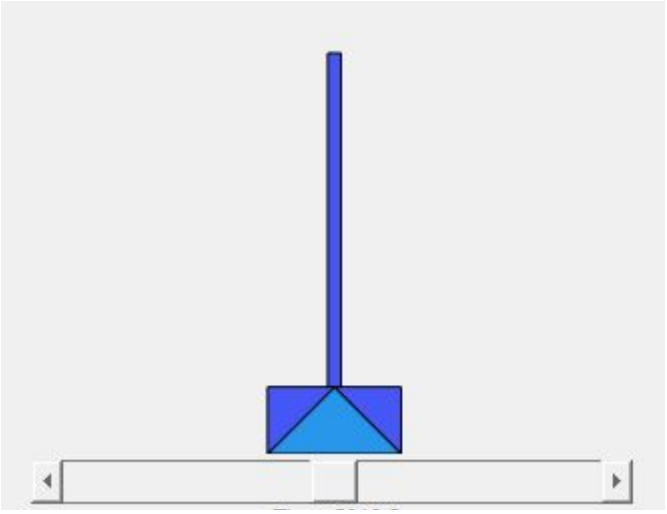
Nous pouvons observer 3 différents graphiques dans cette première capture. Tout d'abord, le premier graphique représente le signal de contrôle en fonction du temps. Nous pouvons remarquer que ce dernier est échantillonné avec une période de 0,01s. Nous n’observons plus de variations à partir d’environ 3s.
Ensuite, le second graphique représente la position angulaire du pendule en fonction du temps. Nous pouvons observer les mêmes résultats que lors du test précédent, c’est-à-dire avec de fortes variations au démarrage du script, puis une valeur finale atteinte à 0 après environ 5s.
Enfin, le dernier graphique représente la position horizontale du chariot par rapport au temps. Encore une fois, nous pouvons observer les mêmes résultats que lors du test précédent. Nous pouvons observer ce coup-ci une position finale au centre de l’axe horizontal, avec une position finale atteint après 6s encore une fois.



Nous pouvons observer sur ce graphique la planification des tâches en fonction du temps concernant le contrôle. Nous pouvons remarquer 10 activation en 1s du système de contrôle (ou de calcule). C’est en accord avec l'échantillonnage réalisé sur le signal de contrôle.



Sur ce graphique, nous pouvons observer la planification de tâches du signal en fonction du temps. La courbe du haut (en bleu), représente la planification de la tâche de restitution. La courbe du bas (en jaune) représente la planification de la tâche d’acquisition. Ces deux tâches sont gérés par le même module. Nous pouvons remarquer qu’il n’y a jamais de préemptions et qu’elles s'alternent bien l’une après l’autre. La tâche d’acquisition occupe environ 30% du temps, tandis que la tâche de restitution est beaucoup moins dépendante, avec une occupation d’environ 10% du temps imparti.



Simulation 2D du pendule avec modules

On se sert de l'affichage 2D du modèle précédent pour vérifier le bon fonctionnement de ce nouveau modèle. Nous pouvons remarquer qu'il se comporte comme nous le souhaitons. Cela nous permet de vérifier encore une fois que notre dernier modèle est bon et fonctionne correctement.

Maintenant que nous avons un code de contrôleur fonctionnel, nous allons développer les modules permettant de l'implémenter dans une maquette physique.
Nous allons commencer par l'acquisition afin de pouvoir lire les valeurs des capteurs et ainsi obtenir les signaux d'entrée.

5. Module Entrée/Sortie :

c. Développement du module spécifique de gestion entrée

Pour acquérir les données des capteurs, nous utilisons la carte 3718. Nous développons le header module_Entree.h afin d'avoir 3 fonctions disponibles et utilisables par les autres modules :

int init3718(void);
Permet d'initialiser la carte d'acquisition, elle retourne un entier nul quand l'initialisation est correcte.

Nous ne nous servons pas d'interruption ni de handler, la fonction d'initialisation est donc inutile dans ce cas.

void SetChanel(int in_channel);
Cette fonction doit positionner le numéro de canal in_channel sur lequel devra être réalisée une conversion Analogique-Numérique.

Pour définir le channel lu, la documentation indique qu'il faut définir deux numéros de channel définissant l'intervalle des channels qui vont être lus par balayement.

BASE+2 (write) - start and stop scan channels								
Bit	D7	D6	D5	D4	D3	D2	D1	D0
Value	CH3	CH2	CH1	CH0	CL3	CL2	CL1	CL0

CH3 to CH0	Stop scan channel number
CL3 to CL0	Start scan channel number

Comme nous lisons qu'un seul channel, nous indiquons le même numéro correspondant au channel demandé :

```
void SetChanel(int in_channel){
  sel_channel = in_channel;
  outb(in_channel + (in_channel<<4), REG_MUX);
}
```

NB : Le channel REG_MUX correspond à BASE+2.

void ADRangeSelect(int channel, int range);
Sélectionne la sensibilité (range) du canal channel

D'après la documentation, la sensibilité est définie selon le tableau suivant :

BASE+1 (write only) - A/D range control code								
Bit	D7	D6	D5	D4	D3	D2	D1	D0
Value	N/A	N/A	N/A	N/A	G3	G2	G1	G0

PCM-3718H range code:

Input Range(V)	Unipolar/Bipolar	Range Code			
		G3	G2	G1	G0
±5	B	0	0	0	0
±2.5	B	0	0	0	1
±1.25	B	0	0	1	0
±0.625	B	0	0	1	1
0 to 10	U	0	1	0	0
0 to 5	U	0	1	0	1
0 to 2.5	U	0	1	1	0
0 to 1.25	U	0	1	1	1
±10	B	1	0	0	0
N/A		1	0	0	1
N/A		1	0	1	0
N/A		1	0	1	1
N/A		1	1	0	0
N/A		1	1	0	1
N/A		1	1	1	0
N/A		1	1	1	1

Comme nous n'utilisons que des composants fonctionnent en -10/+10 V Bipolaire, nous n'intégrant pas de fonction complexe de paramètre utilisateur : le code doit être directement entrée en paramètre en suivant le tableau.

La documentation précise qu'il faut d'abord sélectionner le channel dont on veut changer la sensibilité, on utilise donc la fonction SetChannel dans ADRangeSelect :

```
void ADRangeSelect(int channel, int range){
  SetChanel(channel);
  outb(range, REG_RANGE); //Range code (nous utilisons +-10Vdonc range = 0x08)
}
```

NB : Le channel REG_RANGE correspond à BASE+1

u16 ReadAD(void);
Réalise la conversion Analogique-Numérique sur le canal courant et retourne la valeur convertie.

D'après la documentation, le registre STATUS (BASE+8) permet d'obtenir des informations sur le statut de la conversion :

BASE+8 - A/D status								
Bit	D7	D6	D5	D4	D3	D2	D1	D0
Value	EOC	N/A	MUX	INT	CN3	CN2	CN1	CN0

EOC End of Conversion.
- 0 The A/D converter is idle, ready for the next conversion. Data from the previous conversion is available in the A/D data registers.
- 1 The A/D converter is busy, implying that the A/D conversion is in progress.

MUX Single-ended/differential channel indicator.
- 0 8 differential channels
- 1 16 single-ended channels

INT Data valid.
- 0 No A/D conversion has been completed since the last time the INT bit was cleared. Values in the A/D data registers are not valid data.
- 1 The A/D conversion is completed, and converted data is ready. If the INTE bit of the control register (BASE+9) is set, an interrupt signal will be sent to the PC bus through interrupt level IRQn, where n is specified by bits I2, I1 and I0 of the control register. Though the A/D status register is read-only, writing to it with any value clears the INT bit.

CN3 to CN0 When EOC = 0, these status bits contain the channel number of the next channel to be converted.

Les données, une fois traitées, peuvent être récupérées sur le registre BASE+0 et BASE+1 :

BASE+0 (read only) - A/D low byte & channel number								
Bit	D7	D6	D5	D4	D3	D2	D1	D0
Value	AD3	AD2	AD1	AD0	C3	C2	C1	C0

BASE+1 (read only) - A/D high byte								
Bit	D7	D6	D5	D4	D3	D2	D1	D0
Value	AD11	AD10	AD9	AD8	AD7	AD6	AD5	AD4

AD11 to AD0 Analog to digital data. AD0 is the least significant bit (LSB) of the A/D data, and AD11 is the most significant bit (MSB)

C3 to C0 A/D channel number from which the data is derived. C3 is the MSB and C0 is the LSB

Avant chaque acquisition il faut laisser la carte d'acquisition inactive pendant 10ms car elle ne peut pas traiter les données aussi rapidement qu'elles arrivent.
On utilise busy_sleep et non sleep, afin que l'ordonnanceur aussi soit mis en pause. De cette façon il ne peut pas lancer une acquisition provenant d'une autre tâche.

Pour lancer la conversion, la documentation indique que sans interruption ni autre méthode périodique, il suffit d'écrire n'importe quelle valeur dans le registre de lecture afin de lancer l'acquisition.

Ensuite, on attend que la conversion ne soit plus en cours, soit EOC du registre STATUS passe à 0.

On peut ainsi récupérer les bits de poids faibles sur BASE et ceux de poids fort sur BASE+1. Pour obtenir le résultat il suffit de les additionner en prenant en compte un décalage de 4 à gauche des bits de poids forts.

```
u16 ReadAD(void){
  //printk("-Ch%u Debut read\n", sel_channel);

  rt_busy_sleep(nano2count(PERIODE_CONTROL/2)); //wait 10ms

  outb(1, BASE);//Lance la conversion

  while((inb(REG_STATUS) && 0x80) == 0){ //La conversion est en cours (EOC = 0)
  }

  int lowbyte = inb(BASE)>>4;
  int highbyte = inb(REG_RANGE);
  u16 resultat = lowbyte;
  resultat = resultat + (highbyte<<4);

  int channelLu = inb(BASE) && 0x0F;
  if(channelLu != sel_channel){ //Verification qu'on a bien lu le bon channel
        //printk("- SUPERPOSITION DES TACHES Channel %u a vu %u\n", sel_channel, channelLu);
  }

  outb(0xFF, REG_STATUS);//clear INT bit en écrivant une valeur dedans

  //printk("-Ch%u Fin read\n", sel_channel);

  return resultat;

}
```

Le registre status permet de voir la provenance de l'information lue, soit sont channel. On s'en sert afin de vérifier si les informations proviennent bien du channel attendu, sinon on ne les prend pas en compte.

La documentation conseil de réinitialiser le bit INT du registre status à la fin de l'acquisition.

On renvoie enfin la valeur lue en format u16.

d. Développement du module spécifique de gestion sortie

Pour gérer la sortie de la donnée, nous utilisons la carte 3712. Pour pouvoir l'utiliser, il est nécessaire de l'initialiser correctement en fonction des configurations jumpers qui nous sont imposées, c'est-à-dire en mode asynchrone avec pour le channel 1 et 2 un courant bipolaire à +/- 10V.

void init3712(void);
Permet d'initialiser la carte de restitution. Cette fonction ne retourne rien et n'a pas de paramètres en entrée.

En ce qui concerne l'initialisation de la carte, il est nécessaire d'indiquer qu'une seule chose. Il faut activer le bit de contrôle de la carte. Cela est faisable en appliquant la valeur 1 au registre BASE + 5 (sachant que l'adresse de BASE à pour valeur 0x300). Pour cela on applique la valeur 128 à ce registre.

Write	Output Control							
Bit #	7	6	5	4	3	2	1	0
BASE + 05H	ZD	X	X	X	X	X	X	X

Table C-7 Register for output control

void setDA_async(int channel, int value);
Cette fonction permet de définir quel channel appliquer la valeur en sortie. Elle prend en entrée le channel sur lequel appliquer la valeur souhaitée, et la valeur à appliquer.

Lors de l'utilisation de cette fonction, nous utilisons 2 fonctions qui nous ont été fournies déjà compilée, qui sont **PCM3712setda0(value);** et **PCM3712setda1(value);**. Pour pouvoir utiliser ces fonctions, il faut importer dans notre module de sortie le script **3712.h** qui nous est fourni. Pour ce faire :

```
10   #include"3712.h"
```

A présent, il ne nous reste plus qu'à écrire la fonction qui nous permet d'associer la valeur souhaitée au bon channel.

```
void setDA_async(int channel, int value){
  if(channel == 0){
    PCM3712setda0(value);
  }
  else{
    PCM3712setda1(value);
  }
  outb(1,PCM3712_SYNC);
}
```

e. Test du module de gestion Entrée/Sortie

Maintenant que nos 2 modules sont écrits, il faut être capable de les tester pour vérifier leur fonctionnement.
Pour commencer, nous souhaitons être capable de tester nos modules dans un autre module. Il faut donc créer les script **module_Entree.h** et **module_Sortie.h**. Pour cela nous avons repris l'exemple que nous avions avec **3712.h**. Il ne faut pas oublier d'exporter les fonctions que nous souhaitons utiliser à la fin du script de **module_Entree.h** et **module_Sortie.h**.

Une fois que ces script sont écrit et que les fonctions sont bien exportées, il ne nous reste plus qu'à importer **module_Entree.h** et **module_Sortie.h** dans notre module de test et d'écrire une fonction nous permettant de tester le bon fonctionnement de nos modules entrée/sortie.

void test_acq_c1(int id);
Cette fonction nous permet de tester le bon fonctionnement de nos modules.

Pour mieux comprendre comment nous avons testé ces derniers, nous nous y sommes pris de cette façon :

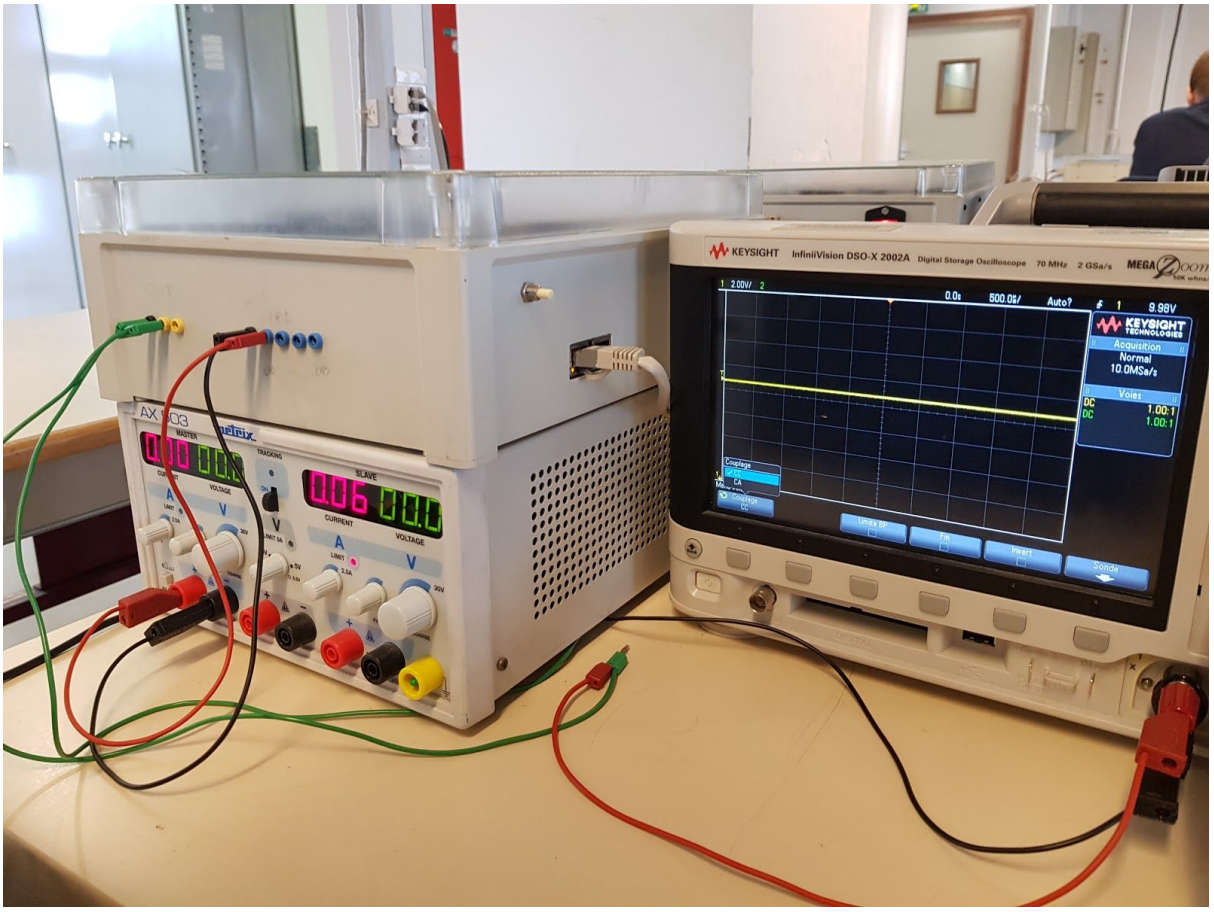
```
u16 value1, value2;
while(1){
	ADRangeSelect(0, 8);
	value1 = ReadAD();

	setDA_async(0, value1);

	ADRangeSelect(1, 8);
	value2 = ReadAD();

	setDA_async(1, value2);

	rt_task_wait_period();
}
```

Nous créons une boucle infinie où nous appliquons le range souhaité sur le channel, puis nous récupérons la valeur lue à la conversion de **ReadAD();**, enfin on applique au channel 0 cette valeur lue. Nous faisons de même pour le second channel.
Il est très important de choisir les valeurs 0 et 1 pour les channels, sinon le programme ne fonctionnera pas. Nous avons mis plusieurs jours à comprendre pourquoi nos programmes ne retournaient pas une bonne valeur car nous avions mis comme valeur de channels 1 et 2.
Il faut aussi appliquer des variables locales au test, sinon cela ne fonctionne pas.

Enfin il ne nous reste plus qu'à créer une tâche. Cette dernière s'exécute toute les 20ms de manière périodique.

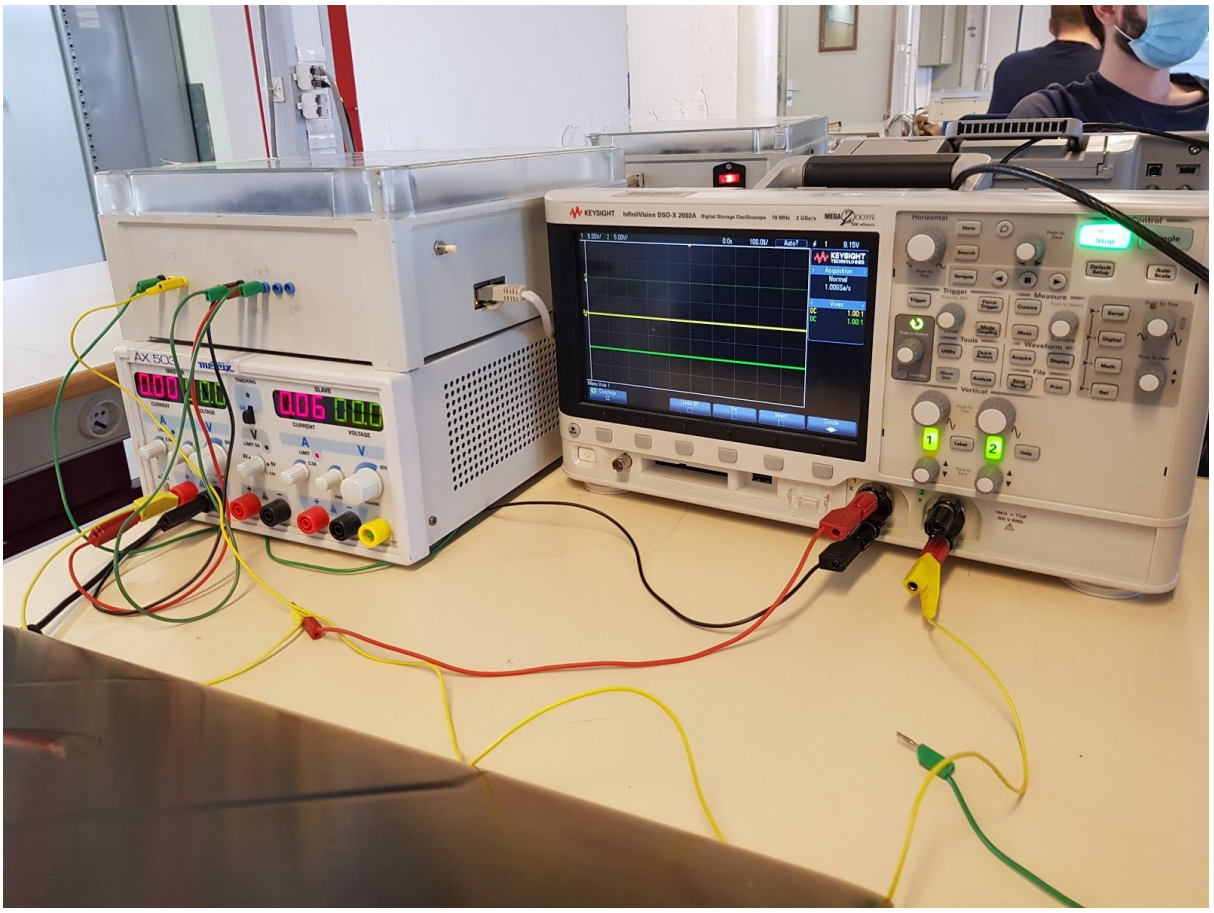
```
rt_set_oneshot_mode();
ierr = rt_task_init(&task_acq_c1, test_acq_c1, 0, STACK_SIZE, PRIORITE, 0, 0);
start_rt_timer(nano2count(TICK_PERIOD));
now = rt_get_time();
rt_task_make_periodic(&task_acq_c1, now, nano2count(PERIODE_CONTROL));
```

Suite à nos tests, nous pouvons observer sur l'oscilloscope le résultat suivant lorsque nous vérifions le bon fonctionnement d'un seul channel à la fois :



Nous pouvons observer que lorsque nous appliquons une entrée à 0V, nous observons bien une sortie à 0V sur l’oscilloscope (sur la photo le calibrage de l’oscilloscope n’est pas parfait d'où le léger décalage)

Nous avons ensuit testé avec 2 entrées :



Nous observons encore une fois un bon résultat à la suite aux entrées.

Enfin, nous nous connectons à la carte pour observer ce qu’elle lit via une requête **dmesg**. Cela nous a permis de vérifier une fois de plus que nos valeurs sont bonnes et surtout celles attendues.

Nous avons essayé de mettre en place des interruptions pour contrôler l'acquisition. Cela fonctionne correctement sur le programme final avec 1 seule tâche (une seule donnée en acquisition). Cependant nous avons observé des problèmes de recouvrement avec 2.
Nous avons implémenté des sémaphores afin de résoudre le problème. mais nous sommes revenu sur la technique de boucle while(STATUS) pour des raisons de simplicité.
Le code est toujours disponible en commentaire sur nos sources github.

6. Module Acquisition/Restitution :

Étant donné leurs similarité, nous décidons de combiner ces deux fonctionnalités en un module, il s’agit de spécifier l'acquisition et la restitution des modules Entrée et Sortie à notre système : channel 0 pour la lecture de l’angle et channel 1 pour la position, les deux en sensibilité -10/+10 V bipolaire, et la sortie de la commande sur le channel 0 des sorties.

a. Développement de l'acquisition

Pour des raisons de gestion de mémoire sur ARCOM, nous devons définir quel espace en mémoire stock les données des valeurs lues. Pour cela nous utilisons des pointeurs dans le fichier header du module.

```
u16 ptr_angle;
u16 ptr_pos;

u16 getAngle(u16 * ptr_angle);
u16 getPos(u16 * ptr_pos);
```

ptr_angle et ptr_pos sont les variables associées aux valeurs lues. Pour y accéder on utilise des getters demandant le pointeur ou se trouve l'adresse en mémoire de la variable.

On définit ces méthode dans le module :

```
u16 getAngle(u16 * ptr_angle){
  *ptr_angle = value0;
  return *ptr_angle;
}

u16 getPos(u16 * ptr_pos){
  *ptr_pos = value1;
  return *ptr_pos;
}
```

On attribue bien les valeurs lues (value0 et value1 locale) aux valeurs des pointeurs.

Dans le module même, on définit les 2 acquisitions dans la même tâche pour simplifier le problème :

```
//taches
rt_set_oneshot_mode();
ierr = rt_task_init(&task_acq, methode_acq, 0, STACK_SIZE, PRIORITE, 1, 0);

start_rt_timer(nano2count(TICK_PERIOD));
now = rt_get_time();
rt_task_make_periodic(&task_acq, now, nano2count(PERIODE_CONTROL/2));
```

On définit bien la période comme la période d'échantillonnage de 10 ms.

Dans l'initialisation on définit les sensibilité -10/+10 au deux channels :

```
printk("Init Module Acquisition Restitution (AR)\n");
ADRangeSelect(0, 8);
ADRangeSelect(1, 8);
```

Les variables locales récupèrent leur valeur ainsi :

```
u16 value0, value1;

void methode_acq(int id){ //acquisition
  while(1){
        printk("-------------\n");

        //angle sur channel 0
        SetChanel(0);
        value0 = ReadAD();
        //printk("Resultat Angle (0/4096) : %u\n", value0);

        //FIFO
        char * tampon;
        tampon = value0;
        rtf_put(FIFO, tampon, sizeof(char));

        //wait
        rt_busy_sleep(nano2count(TICK_PERIOD*5));

        //position sur channel 1
        SetChanel(1);
        value1 = ReadAD();
        //printk("Resultat Position (0/4096) : %u\n", value1);


        rt_task_wait_period();
  }
}
```

L'angle est définie sur le channel 0 et la position sur le 1. On attend 5ms entre les deux acquisitions afin d'éviter toute superposition des valeurs.
La FIFO n'est pas utilisée mais permet d'afficher les valeurs obtenues après le test pour effectuer un diagnostic.

b. Développement de la restitution

La restitution étant réalisée après la tâche de calcul de contrôle, il suffit de la lancer après celle-ci. On n'as pas à définir de tâche ici, il suffit donc de déclarer une méthode appelant le module Sortie :

```
void setCmd(u16 value){
  setDA_async(0, value);
}
```

En spécifiant le channel de sortie 0 pour la commande.

c. Test du module

Comme pour les modules entrées/sorties, nous souhaitons tester notre module Acquisition/réception sur un autre module de test. Pour cela, il faut encore une fois passer par un script **module_AR.h**. Il faut aussi encore une fois ne pas oublier d'exporter les fonctions à la fin du script **module_AR.c** (pour plus de détailles, toutes les explications de cette étapes sont données dans la partie "Test du module de gestion Entrée/Sortie").

Ce module va nous permettre de tester notre modèle avec les vrais valeurs. C'est-à-dire les valeurs d'angle et de position de notre pendule.

La première chose à faire est de calibrer notre programme en fonction du pendule. En effet, chaque pendule est différent et il faut donc faire une phase de calibration avant toute chose. Pour ce faire, nous déplaçons la pendule aux extrémités du rail en position max et min et nous récupérons les valeurs. Nous faisons de même avec l'angle max et min que peut prendre le pendule. Il nous reste une dernière valeur à calibrer qui est celle du centre du pendule. Théoriquement cette dernière valeur sera toujours la même (c'est-à-dire lorsque la valeur retournée est 2048). A terme, cette étape de calibration se serait faite toute seule avec un programme qui aurait permis de faire cela (piste d'amélioration).
Lors de notre dernier test, voici les valeurs que nous avions obtenus (en V) :

```
float angle_MAX = 3.620;
float angle_MIN = -3.519;

float position_MAX = 8.413;
float position_MIN = -8.373;
```

Pour tester notre module, nous avons encore une fois créer une boule infini comme suit :

```
while(1){

        value0 = getAngle();
        value1 = getPos();

        valueVolt0 = (value0-2048)*10/4095;
        valueVolt1 = (value1-2048)*10/4095;

        printk("Resultat Angle Volt (-10/+10): %u\n", valueVolt0);
        printk("Resultat Position Volt (-10/+10): %u\n", valueVolt1);

        valueAngle = valueVolt0*17.13/3.620;
        valuePos = valueVolt1*1/8.413;

        //setCmd(1024);

        rt_task_wait_period();
}
```

Ce que nous faisons c'est tout d'abord de récupérer les valeurs d'angles et de positions actuelles de notre pendule.
Ensuite nous appliquons un produit en croix pour convertir et normaliser ces valeurs à +/-10V.

Volt (format binaire)	Volt
value-2048	valueVolt
4095	10

Enfin nous convertissons encore une fois ces valeurs à l'aide d'un produit en croix pour avoir les valeurs en mètres et en degré de ces valeurs.

Volt	Angle (en degré)
valueVolt0	valueAngle
3.620	17.13

Volt	Position (en mètres)
valueVolt1	valuePos
8.413	1

Ces 2 dernières étapes nous permettent uniquement de vérifier que les valeurs en sorties sont bien celles observées en réalité (à l'aide d'un oscilloscope principalement).

Enfin il est nécessaire comme c'est un test de créer une tâche comme la dernière fois. Nous avons donc fait le choix de créer exactement la même tâche que pour le précédent test :

```
rt_set_oneshot_mode();
ierr = rt_task_init(&task_acq_c1, test_acq_c1, 0, STACK_SIZE, PRIORITE, 0, 0);
start_rt_timer(nano2count(TICK_PERIOD));
now = rt_get_time();
rt_task_make_periodic(&task_acq_c1, now, nano2count(PERIODE_CONTROL));
```

7. Contrôleur :

a. Développement du module de contrôle

Le développement du contrôleur revient à traduire le contrôleur simulé sous matlab en C :

```
//matrice de modélisation du systeme
float adc[4][4] = {{0.6300f, -0.1206f, -0.0008f, 0.0086f}, {-0.0953f, 0.6935f, 0.0107f, 0.0012f}, {-0.2896f, -1.9184f, 1.1306f, 0.2351f}, {-3.9680f, -1.7733f, -0.1546f, 0.7222f}};
float bdc[4][2] = {{0.3658f, 0.1200f}, {0.0993f, 0.3070f}, {1.0887f, 2.0141f}, {3.1377f, 1.6599f}};
float cdc[4] = {-80.3092f, -9.6237f, -14.1215f, -23.6260f};
float ddc[2] = {0.0f, 0.0f};

//vecteur d'état
float x[4] = {0.0f, 0.0f, 0.0f, 0.0f};
//vecteur sortie
float y[2] = {0.0f, 0.0f};
//vecteur commande
float u = 0.0f;
```

On définit donc les paramètres du système déjà calculés : matrices, vecteur d'état, sortie et commande.

```
void methode_ctrl(int id){ //tâche controleur
  u16 cmde = 0;
  u16 value0, value1;
  float valueVolt0, valueVolt1;
  float valueNorm0, valueNorm1;
  float temp[4] = {0.0f, 0.0f, 0.0f, 0.0f};
  while(1){
        // Code iteratif du controle

        // Acquisition angle/position
        value0 = getAngle(&ptr_angle);
        value1 = getPos(&ptr_pos);

        valueNorm0 = (value0-2036)*0.298974/(2792-2036);              //17.13 °
        valueNorm1 = (value1-2162)*0.91/(3880-2162);

        y[0] = valueNorm0;
        y[1] = valueNorm1;

        // Calcul vecteur etat et commande
        temp[0] = adc[0][0]*x[0] + adc[0][1]*x[1] + adc[0][2]*x[2] + adc[0][3]*x[3] + bdc[0][0]*y[0] + bdc[0][1]*y[1];
        temp[1] = adc[1][0]*x[0] + adc[1][1]*x[1] + adc[1][2]*x[2] + adc[1][3]*x[3] + bdc[1][0]*y[0] + bdc[1][1]*y[1];
        temp[2] = adc[2][0]*x[0] + adc[2][1]*x[1] + adc[2][2]*x[2] + adc[2][3]*x[3] + bdc[2][0]*y[0] + bdc[2][1]*y[1];
        temp[3] = adc[3][0]*x[0] + adc[3][1]*x[1] + adc[3][2]*x[2] + adc[3][3]*x[3] + bdc[3][0]*y[0] + bdc[3][1]*y[1];
        x[0] = temp[0];
        x[1] = temp[1];
        x[2] = temp[2];
        x[3] = temp[3];
        u = 0.9*cdc[0]*x[0] + 0.9*cdc[1]*x[1] + 0.9*cdc[2]*x[2] + 0.9*cdc[3]*x[3]; //1.4* = gain

        // Convertion en binaire
        if(u >= 10){
                u = 10;
        }else if(u <= -10){
                u = -10;
        }
        cmde = (u+10)*(-4095)/20;

        // Envoi commande
        setCmd(cmde);

        rt_task_wait_period();
  }
}
```

Avant d'effectuer les calculs matriciels comme dans la simulation, il faut ramener les entrées à la même unité.
Pour cela on mesure la longueur du chariot du pendule de 0.91 mètre avec une valeur centrée de 2162 lue par le capteur. On mesure aussi un angle max de 17.13° soit 0.2989 radian avec une valeur lue par le capteur de 2036 à l'angle 0.

value0 Volt (format binaire)	valueNorm0 Radian (convertie)
2036	0°
2792	17.13° soit 0.2989 rad (max)

value1 Volt (format binaire)	valueNorm1 Mètre (convertie)
2162	0 m
3880	0.91 m (max)

On effectue alors 2 produits en croix en supprimant bien la valeur en zéro afin de centrer correctement les valeurs. (les valeurs en volt sont définies sur 0 - 4095)

Pour ne pas endommager le servomoteur, on vérifie que la tension envoyée est bornée en -10/+10 avec une condition simple.
Ensuite on convertit la commande en volt (format binaire) par un produit en croix. on ajuste le gain de 0.6 à 1.4 (0.9 plus efficace) avec un -1 comme dans la simulation.

b. Test du module de contrôle

Pour tester le module de contrôle, on affiche avec les printk les valeurs aux différentes étapes du calcul, on s'assure des ordres de grandeurs et de l'évolution des valeurs lorsqu'on bouge le chariot d'un côté ou de l'autre, la commande doit changer de signe lorsqu'on change l'angle du pendule d'un côté ou de l'autre de zéro.
On peut ensuite brancher la puissance du servomoteur.

c. Test sur maquette et optimisations potentielles & problèmes

Il est à présent temps de mettre tous nos modules en commun et de tester sur le modèle réel. Malheureusement nous n'avons pas de photos du bon fonctionnement de notre pendul mais voici ce qui se passait:

- Si nous positionnons notre pendule à la verticale et au centre du rail, notre pendule reste stable et oscille plus ou moins en fonction des perturbations et du gain que nous avons entré.
- Si nous appliquons le même test mais légèrement décalé de sa position centrale, le pendule va s'équilibrer et se déplacer vers le centre du rail.
- Si mettons le pendule à l'extrémité du rail avec la tige en position libre, alors le pendule n'arrive pas à s'équilibrer. Cela reste encore à travailler et optimiser la vitesse du code en trouvant le gain exact pour notre pendule.

8. Module Communication :

A cause de la crise sanitaire, nous n'avons pas eu le matériel à disposition afin de réaliser le module de communication bus CAN. Nous ne pouvons donc pas réaliser l'architecture à 2 pendules croisées.