



CentraleSupélec

---

# **RAPPORT TL SIR COMMANDE DE ROBOT PAR INTERFACE CÉRÉBRALE**

**- Rafael ELLER CRUZ, Mathieu DAVIET, Heloise HUYGHUES  
DESPOINTES-**

promo 2018

---

# Table des matières

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Extraction de la commande du robot à partir des signaux cérébraux</b>	<b>2</b>
2.1	Le signal d'entrée . . . . .	2
2.2	Architecture globale de la solution choisie . . . . .	4
<b>3</b>	<b>Contrôle du Robot</b>	<b>7</b>
3.1	Choix d'architecture . . . . .	7
3.2	Mise en oeuvre du système et résultats . . . . .	8
<b>4</b>	<b>Conclusion</b>	<b>9</b>
<b>5</b>	<b>Annexe</b>	<b>10</b>

# 1 Introduction

Lors de ses 11 séances de TL d'approfondissement, nous avons eu l'occasion de réaliser la commande du robot Khepera à partir de signaux provenant d'une interface cerveau machine (BCI). Ce projet nous a permis de mettre en pratique les connaissances que nous avons acquises lors des cours de la majeure Systèmes Interactifs et Robotiques et plus particulièrement celles des cours Modélisation et Analyse Spectrale, Robotique autonome et Programmation C++.

Deux robots Khepera, un système d'acquisition cérébrale et des signaux pré-enregistrés ont été mis à notre disposition pour ce projet. Le robot ne sera commandé que par 4 ordres: rester immobile, tourner à droite, tourner à gauche et avancer.

Notre travail à travers les séances et dans ce rapport est décomposé en 2 grandes étapes: l'extraction des commandes sur les signaux BCI et le contrôle du robot. Ces deux étapes sont ensuite réunies sous C++ grâce au framework ROS.

## 2 Extraction de la commande du robot à partir des signaux cérébraux

### 2.1 Le signal d'entrée

Le signal de commande que l'on va être amené à analyser en entrée du système correspond au signal en sortie de la BCI. Dans le cadre de ce TL, 3 signaux ont été enregistrés à une fréquence de 256Hz et peuvent être rejoués à volonté. Pour chaque période de temps, les signaux enregistrés possèdent deux valeurs correspondant aux deux canaux du capteurs. Nous aurions pu effectuer nos propres enregistrements à l'aide d'un capteur BIOSEMI à 2048Hz mais par manque de temps, nous nous sommes concentrés sur les signaux déjà enregistrés.

Le signal en sortie de la BCI est influencé par trois plaques clignotant respectivement autour de 7.5Hz, 11Hz et 13.5Hz. Lorsque l'expérimentateur regarde l'une des plaques à une fréquence donnée, cette fréquence se répercute dans l'émission de ses signaux cérébraux et se retrouve dans le signal récupéré par la BCI. Chacune de ces plaques clignotantes correspond à un ordre particulier. L'expérimentateur doit regarder la plaque clignotant à 7.5Hz si il veut que le robot tourne à gauche, celle clignotant à 11Hz si il veut que le robot avance, celle clignotant à 13.5Hz si il veut que le robot tourne à droite et ne regarder aucune plaque si il veut que le robot s'arrête.

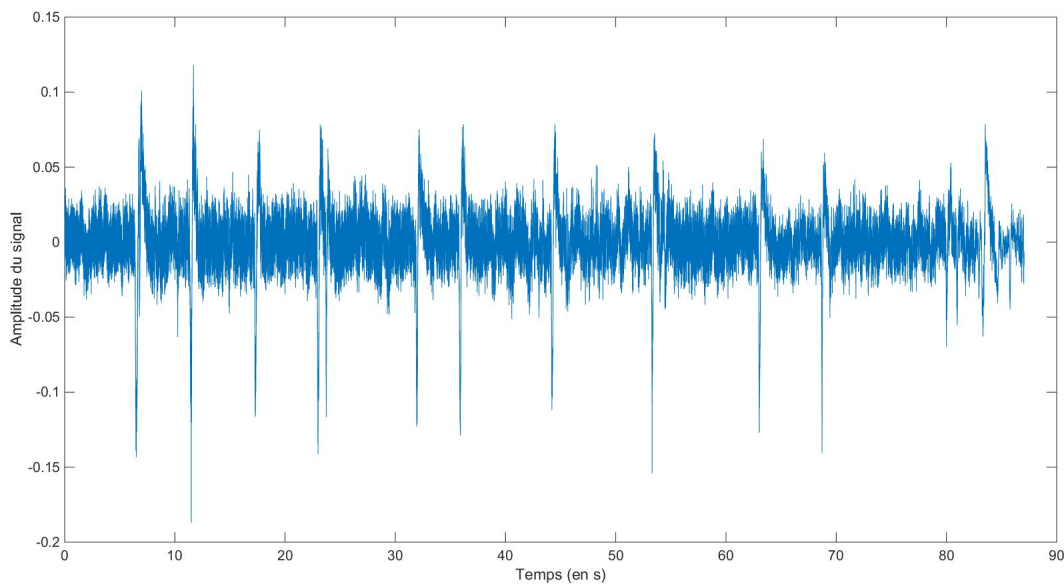


Figure 1: Signal brut en sortie de la BCI enregistré

Pour faciliter l'exploitation des résultats nous avons appliqué deux modifications aux signaux enregistrés.

Tout d'abord, les signaux enregistrés sont bi-canaux. Ainsi, pour qu'ils soient plus faciles à manipuler par la suite, nous les avons transformé en signaux mono-canal correspondant à la moyenne des deux canaux.

Ensuite, les signaux enregistrés ne possèdent que les valeurs du capteur en fonction du temps, les labels indiquant quels ordre ont été donné dans le temps ne sont disponibles que de manière écrite

dans le sujet. Pour faciliter la modélisation et l'optimisation de la commande de robot, nous avons ajouté une deuxième colonne au signaux mono-canal enregistrés correspondant à l'ordre attendu en fonction du temps. Pour ce faire, nous avons associé un nombre à chaque ordre: 0 pour l'ordre ne rien faire, 1 pour l'ordre d'avancer, 2 pour l'ordre de tourner à droite et 3 pour l'ordre de tourner à gauche. L'ajout du label automatiquement nous permettra par la suite de créer des scores pour évaluer l'efficacité du traitement du signal que nous effectuerons.

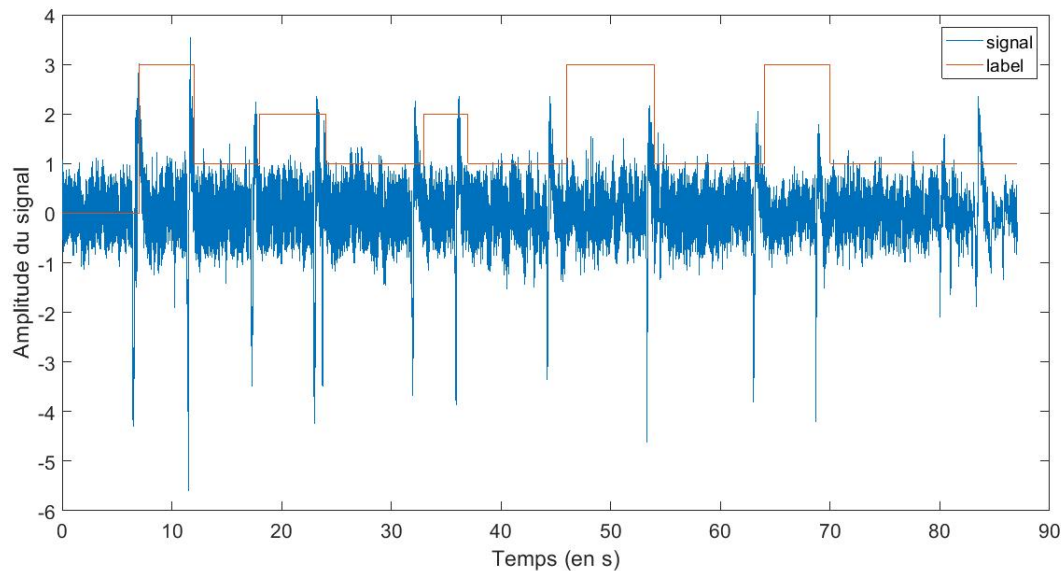


Figure 2: Signal brut monocanal labellisé

## 2.2 Architecture globale de la solution choisie

Comme on peut le constater sur la figure 1, le signal a besoin d'être traité pour pouvoir être exploité par la suite. La première étape va consister à extraire du signal de départ 3 signaux autour des fréquences qui nous intéressent à savoir 7.5 Hz, 11Hz et 13.5Hz. Pour effectuer cela nous utilisons 3 filtres passe-bande dont les paramètres sont la bande passante:  $\Delta f$ , le gain:  $G$  et la fréquence centrale:  $f_c$ . La fréquence centrale correspond respectivement à 7.5Hz, 11Hz et 13.5Hz pour chacun des 3 filtres. Nous verrons par la suite le moyen par lequel on a optimisé les autres paramètres. Comme le montre la figure 3 ci-dessous, nous pouvons constater que ce premier filtre nous permet déjà de distinguer les ordres donnés par l'expérimentateur via la BCI.

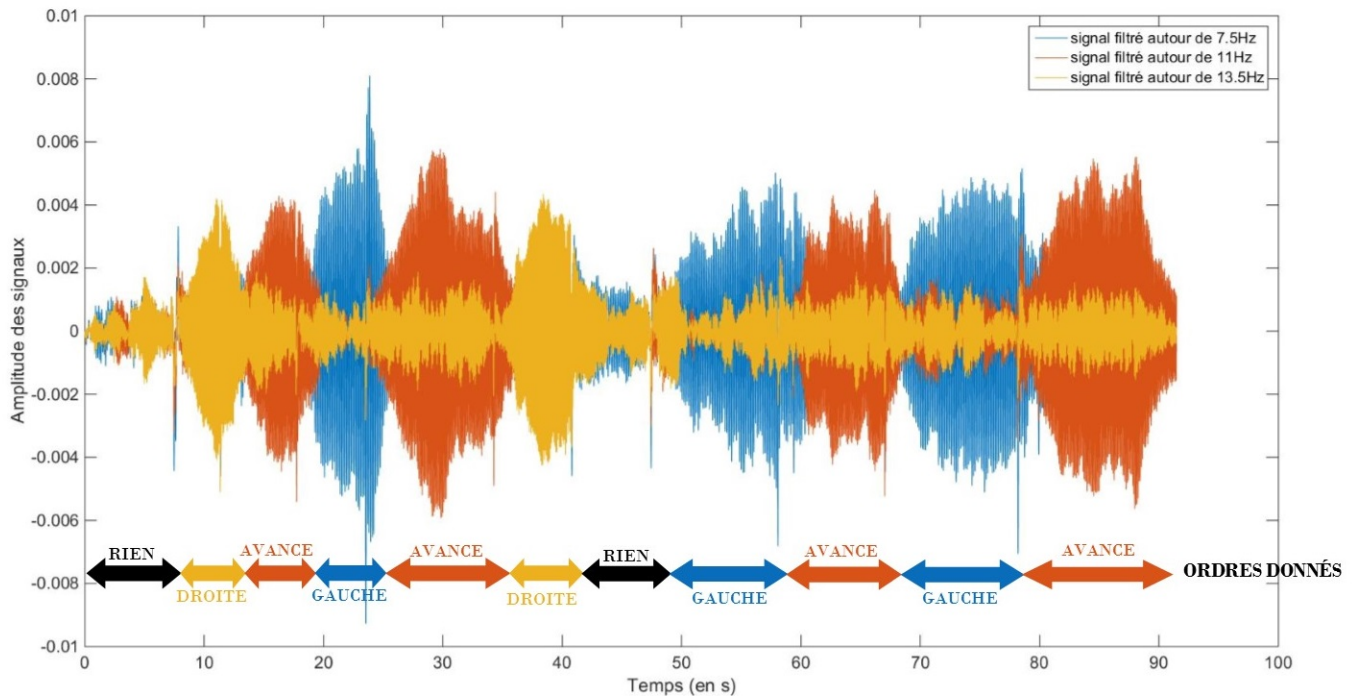


Figure 3: Trois signaux en sortie du filtre passe bande centrés respectivement en 7.5Hz, 11Hz et 13.5Hz

Malgré que le fait que l'on arrive à distinguer à l'œil nu les ordres à partir des 3 signaux en sortie des filtres passe-bande, il est nécessaire d'obtenir des signaux plus propres pour par la suite formaliser un processus de décision concernant la commande du robot. Pour cela nous allons calculer la puissance lissée de chacun des 3 signaux.

L'effet de lissage permet de réduire le bruit mais cela augmente l'inertie et diminue la résolution. Cet effet de lissage dispose d'un paramètre  $\alpha$  dont nous justifierons le choix par la suite.

La puissance lissée nous permet ainsi d'obtenir des signaux exploitables par la suite pour commander le robot grâce à un processus de décision qui va être détaillé dans la suite du rapport.

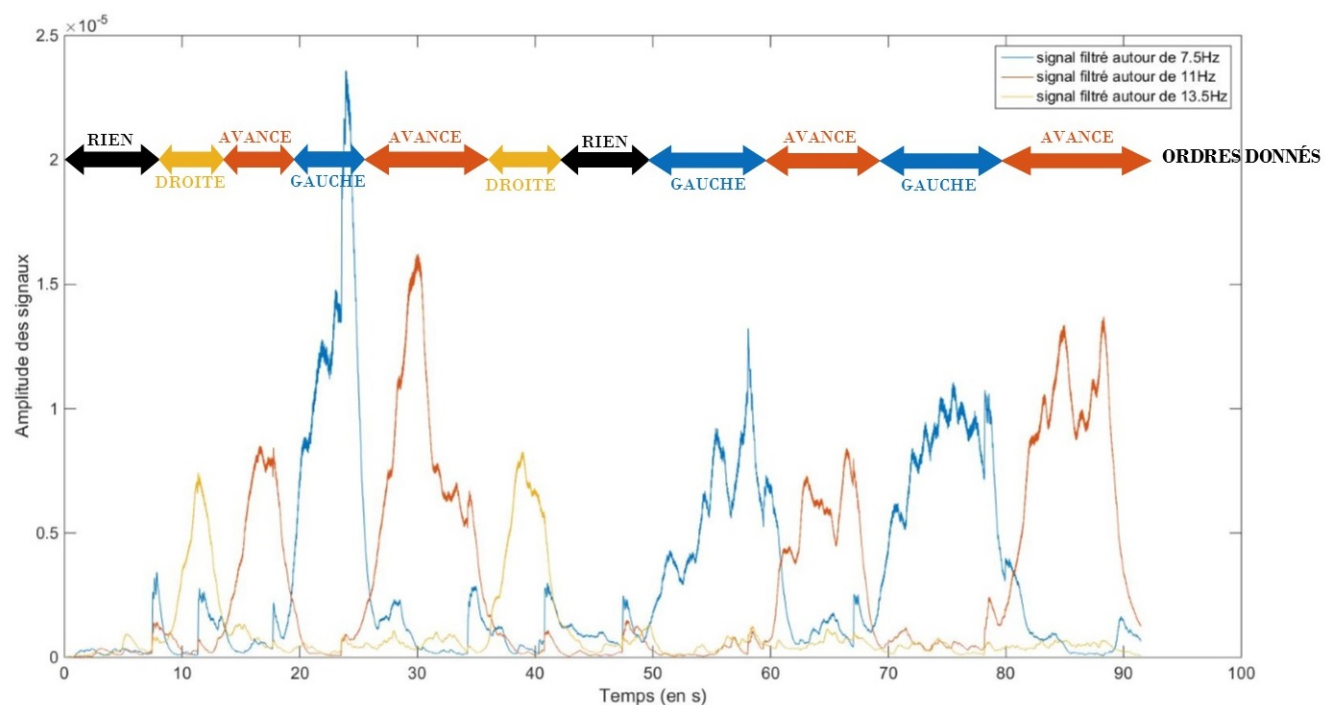


Figure 4: Trois signaux en sortie du filtre passe bande centrés respectivement en 7.5Hz, 11Hz et 13.5Hz

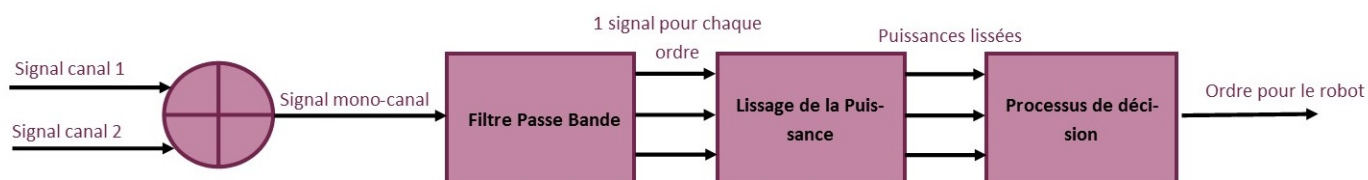


Figure 5: Récapitulatif de l'architecture nous permettant de prendre une décision quant à l'ordre à donner au robot

test1



### 3 Contrôle du Robot

Dans cette section nous discuterons les choix d'architecture, la mise en oeuvre des packages ROS que nous ferons tourner sur le robot et les résultats obtenus. Les indications de comment exécuter le code se trouvent dans la subsection 3.2

#### 3.1 Choix d'architecture

Nous avons choisi de travailler avec ROS (Robot Operating System), un middleware de robotique conçu pour aider à abstraire les détails de hardware, pour qu'on puisse accéder aux fonctionnalités de haut niveau des robots d'une forme simple. Le fonctionnement de ROS est basé sur l'exposition des ressources du système parmi des **noeuds** qui peuvent se communiquer entre eux parmi des **messages** qui sont envoyés à des **topiques**, entités responsables de recevoir les messages des noeuds appelés **publishers** et les envoyer aux noeuds appelés **subscribers**. Les ressources incluent les moteurs et les capteurs, bien que les ressources de computation. Avec ces concepts basiques de ROS, nous allons présenter l'architecture du système, qui est résumé dans la figure suivante.

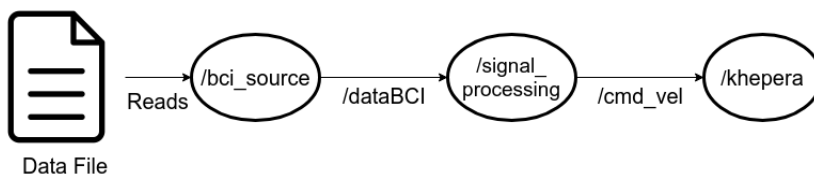


Figure 6: Diagramme conceptuel du système, où les ellipses représentent des noeuds et les flèches représentent des topiques (sauf la flèche 'Reads')

Pour simuler l'acquisition en temps réel des signaux BCI on a décidé de faire tourner un noeud appelé `/bci_source` qui au moment de l'initialisation lit un fichier contenant un vrai signal pré-enregistré (fourni au début du projet) et après il rentre dans une boucle d'exécution, publiant un message sur le topique `/dataBCI` à la fois. Chaque message s'agit d'un float qui correspond à un échantillon du signal. Pour garantir que la simulation soit bien adaptée, ce noeud tourne à une fréquence égale à la fréquence d'échantillonnage du signal, dans ce cas 256 Hz. Le fichier contient aussi les labels des commandes correspondantes qui sont affichés dans l'écran lors de l'exécution du noeud pour que l'utilisateur puisse avoir au moins une idée du comportement du système. Le fichier correspondant à ce noeud se trouve sur `'/projectRoot/ros_ws/bci_robot/scripts/bci_source.py'`.

Le noeud `/signal_processing` est un *subscriber* du topique `/dataBCI`, il reçoit des échantillons à 256 Hz. Différemment de l'approche qu'on a pu tenir dans les sections précédentes, ici nous sommes obligés de traiter les échantillons tant qu'ils viennent, et pas dans un coup du début à la fin. Après quelques discussions dans le groupe et avec les encadrants nous avons décidé filtrer le signal tant qu'il vient et stocker les estimations de puissance en chaque fréquence centrale dans une espèce de buffer de taille fixe, et une fois que ce buffer est rempli nous publions un message de commande au robot correspondant à une moyenne des signaux dans ce buffer pour chaque fréquence. Les calculs faites lors du traitement du signal sont les mêmes que ceux décrits précédemment, c'est à dire, on passe le signal  $x[n]$  pour trois filtres passe-bandes de fréquences centrales 7.5, 11 et 13.5 Hz, obtenant ainsi trois signaux  $y_i[n]$ . À partir de ces signaux on estime la puissance présente en chaque bande de fréquence, obtenant trois signaux  $z_i[n]$ , et ces  $z$  seront, donc, stockés dans le buffer. La stratégie pour choisir une commande à partir de ces signaux  $z_i[n]$  a été déjà exploré dans la section précédente, mais nous y reviendrons dans la subsection suivante. Finalement, après avoir choisi la bonne commande le noeud `/signal_processing` publie un message du type *Twist* au

robot parmi le topique `/cmd_vel`. La fréquence de publication de ce noeud est directement lié à celle du noeud d'avant, et si  $f_e$  est la fréquence d'échantillonnage du signal original et  $N$  la taille du buffer, la fréquence de publication sera de  $f_e/N$ . Le fichier correspondant à ce noeud se trouve sur `'/projectRoot/ros_ws/bci_robot/scripts/signal_pro.py'`.

Finalement, le noeud du robot khepera sera un *subscriber* du topique `/cmd_vel`, d'où il va recevoir les consignes de vitesse correspondantes. Passons maintenant aux détails de mise en oeuvre.

### 3.2 Mise en oeuvre du système et résultats

Nous avons choisi python comme langage pour implémenter ces deux noeuds, dont les fichiers se trouvent dans le répertoire `'/projectRoot/ros_ws/bci_robot/scripts'`. Pour faciliter le débogage et permettre de tester le programme même sans le robot, on a ajouté une option pour envoyer les messages à un noeud turtlesim au lieu du khepera. On décrira pas à pas comment faire tourner le système tant que nous expliquons les détails de sa mise en oeuvre.

D'abord, il faut *builder* les packages ros. On va faire référence à la racine du projet comme **projet**, donc allez sur `projet/ros_ws` et tapez:

```
$ catkin_make
$ source devel/setup.bash
```

Ouvrez un terminal et faites tourner le roscore:

\$ roscore

Maintenant passons au noeud `/bci_source`. Si vous allez sur *projet/obj* vous verrez tous les enregistrements qui peuvent être utilisés avec ce noeud. Alors, tapez:

```
$ rosrun bci_robot bci_source.py [path to file]
```

Où `[path_to_file]` c'est le chemin complet pour un de ces fichiers sur `projet/obj`. Ouvrons `'herve002_labeled.txt'` par exemple:

```
rafael@rafael-Aspire-E5-575G:~/dev/sir/TL-Approf.-BCI/ros_ws$ rosrn bci_robot bci_source
e.py '/home/rafael/dev/sir/TL-Approf.-BCI/obj/herve002_labeled.txt'
Initializing /bci_source
This node feeds a 256 Hz stream of samples from the BCI measurements, to a topic called dataBCI. The current version simulates the real-time acquisition of the signal by reading a text file containing real measurements.
Stop
Stop
Stop
Stop
Stop
Stop
Stop
Stop
Stop
Stop
Stop
```

Figure 7: Fonctionnement du noeud /bci source

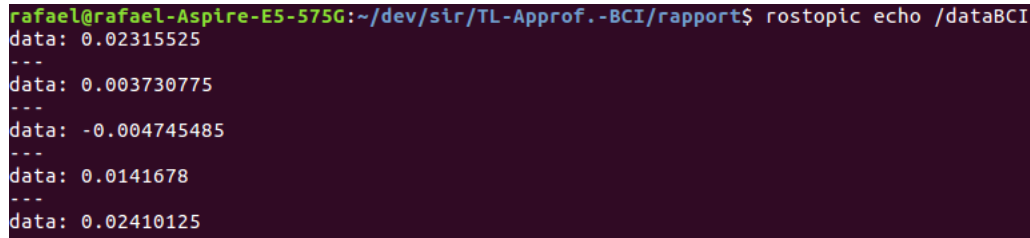
Vous verrez que ce noeud est en train de publier des messages sur le topique `/dataBCI`, tapez:

```
$ rostopic echo /dataBCI
```

Et vous verrez quelque chose similaire à ce qu'on a dans la figure 8

Le noeud est un simple publisher et il continue à publier les échantillons jusqu'à la fin du fichier, et là il s'arrête et termine le programme.

Passons au noeud /signal processing. Ouvrez un nouveau terminal et tapez:



```

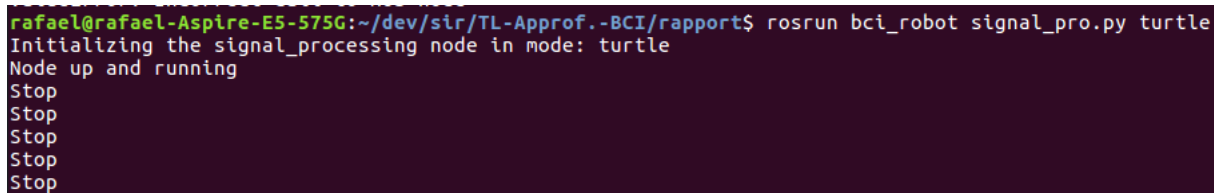
rafael@rafael-Aspire-E5-575G:~/dev/sir/TL-Approf.-BCI/rapport$ rostopic echo /dataBCI
data: 0.02315525
---
data: 0.003730775
---
data: -0.004745485
---
data: 0.0141678
---
data: 0.02410125

```

Figure 8: Publication des échantillons sur /dataBCI

```
$ rosrun bci_robot signal_pro.py turtle
```

Ici nous utilisons l'option 'turtle', parce que nous publierons les commandes dans le topique du turtlesim, pour le faire publier sur le topique du khepera il suffit de changer 'turtle' pour 'khepera'. Si jamais votre noeud /bci\_source a fini de publier tout le signal et a affiché 'Finished publishing the data from file' relancez-le. Vous aurez quelque chose comme la figure 9



```

rafael@rafael-Aspire-E5-575G:~/dev/sir/TL-Approf.-BCI/rapport$ rosrun bci_robot signal_pro.py turtle
Initializing the signal_processing node in mode: turtle
Node up and running
Stop
Stop
Stop
Stop
Stop

```

Figure 9: Initialisation et affichage du noeud /signal\_processing.

Différemment du /bci\_source, ce noeud est implémenté comme une classe appelé **Signal\_processor** (l'autre est défini comme une simple fonction), afin de stocker les paramètres et avoir accès facile à ces variables. L'initialisation de la classe est faite de façon paramétrable, où on peut choisir la taille du buffer, les vitesses du robot, le mode d'opération, bien que mettre en place les paramètres optimaux obtenus auparavant. Ce noeud est aussi un subscriber ainsi qu'un publisher. Il possède une fonction **callback\_data\_listen()** (cf. signal\_pro.py) qui est appelé à chaque fois qu'il reçoit un message du topique /dataBCI, et qui traite les échantillons, un la fois, tant qu'ils arrivent, et ainsi cette fonction est appelé à une fréquence de 256 Hz. Elle a un comportement cyclique dans lequel elle va remplir le buffer à chaque nouvel appel en bougeant un compteur de position en position, mais quand le buffer est plein, cette fonction fait appel à une autre fonction **callback\_pub()** et mets le compteur à zéro. Cette fonction **callback\_pub**, par contre, correspond au côté publisher du noeud, et donc elle va utiliser toute l'information présente dans le buffer pour choisir la bonne commande et la publier sur le topique choisi. La fonction responsable pour trouver cette bonne commande est **get\_command()**. D'abord cette fonction doit utiliser toute l'information dans le buffer, et la façon la plus simple de le faire c'est de faire la moyenne de chacun des 3 signaux dans le buffer. Une approche plus intéressante serait de normaliser ce buffer par une fenêtre (Hamming, Blackman etc) pour diminuer l'effet de bord introduit par la fenêtre rectangulaire du buffer, mais faire juste la moyenne nous rends déjà des résultats intéressants. Après, comme décrit précédemment, la fonction va utiliser 3 seuils obtenues parmi une optimisation en grid search, et va diviser chacun des 3 signaux par le seuil correspondant. De cette façon, quand toutes les signaux sont inférieurs à 1, tous les signaux originales sont en dessous des seuils et ainsi on choisira la commande 'Rien'. L'effet de normaliser les 3 signaux par ces seuils c'est que, une fois que au moins un de ces signaux est au dessus du seuil, il suffit juste de choisir entre eux le plus grand et appliquer la commande correspondante. Les vitesses appliqués par les commandes sont définis par les paramètres

de la classe `Signal_processor`.

Pour voir le fonctionnement des deux noeuds ensemble, faites-les tourner après lancer le noeud `turtlesim`, en tapant:

```
$ rosrun turtlesim turtlesim_node
```

ou en lançant le noeud du `khepera`.

## 4 Conclusion

## 5 Annexe