École de technologie supérieure

Université du Québec

Rapport deuxième étape de projet

Présenté à M. Marcel GABREA

Dans le cadre du cours

*SYS835 – Processeur numérique du signal et ses applications*

PAR

Eric LACERTE LACE23038502

Philippe LAVOIE LAVP05067200

MONTRÉAL, LE 4 DÉCEMBRE 2017

# Introduction

Ce rapport fait suite à la première partie sur l’implémentation de l’algorithme de suppression du bruit de McAulay et Malpass [1]. Ce document explique les méthodes et stratégies utilisées afin d’implémenter une solution en temps réel de ce dernier sur la carte DSK6713. Nous y exposerons les résultats obtenus et feront une analyse de ceux-ci. Nous terminerons avec les conclusions et les recommandations suite à ce projet.

# Implémentation

La solution d’implémentation présentée contient deux projets CSS (.cproj) contenu dans le répertoire *CSS-Projets*.

## Projet sur simulateur

Le projet dans le répertoire *Projet\_SYS835\_test* a été utilisé pour développer l’algorithme de suppression du bruit. Le répertoire *include* contient tous les fichiers d’en-têtes (.h) utilisé pour le projet. On y trouve une description détaillée de chacune des fonctions et variables développées. Nous invitons le lecteur à en prendre connaissance, puisque beaucoup d’informations pertinentes à la compréhension s’y retrouve. Le répertoire *source* contient l’implémentation des diverses fonctions décrites dans les fichiers d’en-têtes.

Afin de faciliter le développement, une cible du simulateur de la carte DSK6713 a été générée pouvoir lancer le code et ainsi vérifier le fonctionnement des fonctions implémentées. Le répertoire *data* contient le fichier de simulation *car.raw[[1]](#footnote-1)* (le même fichier qui a été utilisé pour le développement Matlab) qui est utilisé pour générer les données d’entrée pour la simulation. Ce fichier est l’enregistrement d’une conversation entre un homme (voix plus grave) et une femme (voix plus aigüe) avec la présence de bruits de voitures.

Dans le fichier *main.c*, on génère des échantillons de taille fixe en lisant le fichier *car.raw*  et les traiter en appelant la fonction d’algorithme de suppression du bruit (*NoiseSuppressionAlgorithm*).

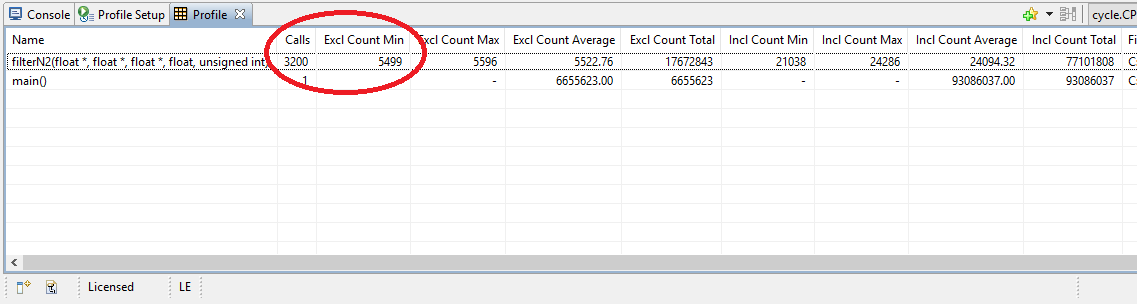
## Projet sur la carte DSK6713

Le projet dans le répertoire Projet\_SYS835 a été utilisé pour vérifier l’algorithme de suppression du bruit en temps réel. Le projet contient pratiquement la même arborescence et les mêmes fichiers que le projet *Projet\_SYS835\_test*. Nous y avons ajouté les fichiers *C6713dskinit.h*, *Vectors\_intr.asm* et *C6713dskinit.c* tirés du fichiers *Ex22b.zip*.[[2]](#footnote-2) Nous nous sommes inspirés de l’exemple *Ex22b.zip* comme point de départ pour l’implémentation en temps réel de notre solution.

Dans le fichier *main.c*, nous configurons le DSP afin d’utiliser les interruptions pour produire des trames de taille fixe et de déclencher le traitement par la fonction NoiseSuppressionAlgorithm. De plus nous avons créé une cible utilisée pour lancer notre implémentation sur la carte DSK6713 et permettre de vérifier notre solution en temps réel.

## Codage en ASM

Un candidat à l’optimisation en assembleur est la fonction implémentant le filtre Butterworth d’ordre 2 (« filterN2 » dans filtre.c). Pour chaque trame de 160 échantillons, cette fonction est appelée 17 fois (pour chaque bande de fréquence). La version en C compilée en Configuration « Release » exécute en 5499 cycles.



Nous avons codé une version du filtre en assembleur («\_FilterASM » dans FilterAsm.asm) dans le but de connaître le type et le nombre d’instructions nécessaires pour une version non-optimisée et d’en extrapoler le gain en performance potentiel après optimisation.

La version en assembleur non-optimisée exécute (23 instructions + 14 nop) \* 160 = 5920 cycles.

Ce qui est presque identique à la performance obtenue avec la version «Release » en C. Donc, en remplaçant les « nop » et en plaçant plusieurs instructions en parallèle, un gain de performance significatif (au moins 40%) serait possible.

# Résultats

## Simulation

À l’aide du simulateur en mode *debug*, nous avons mesuré les différents gains appliqués pour chacun des canaux en différentes circonstances, soit aucune voix n’est présente et lorsque la voix est présente. À noter que sur les graphiques l’axe des X correspond aux canaux et l’axe des Y aux valeurs de gains appliquées.

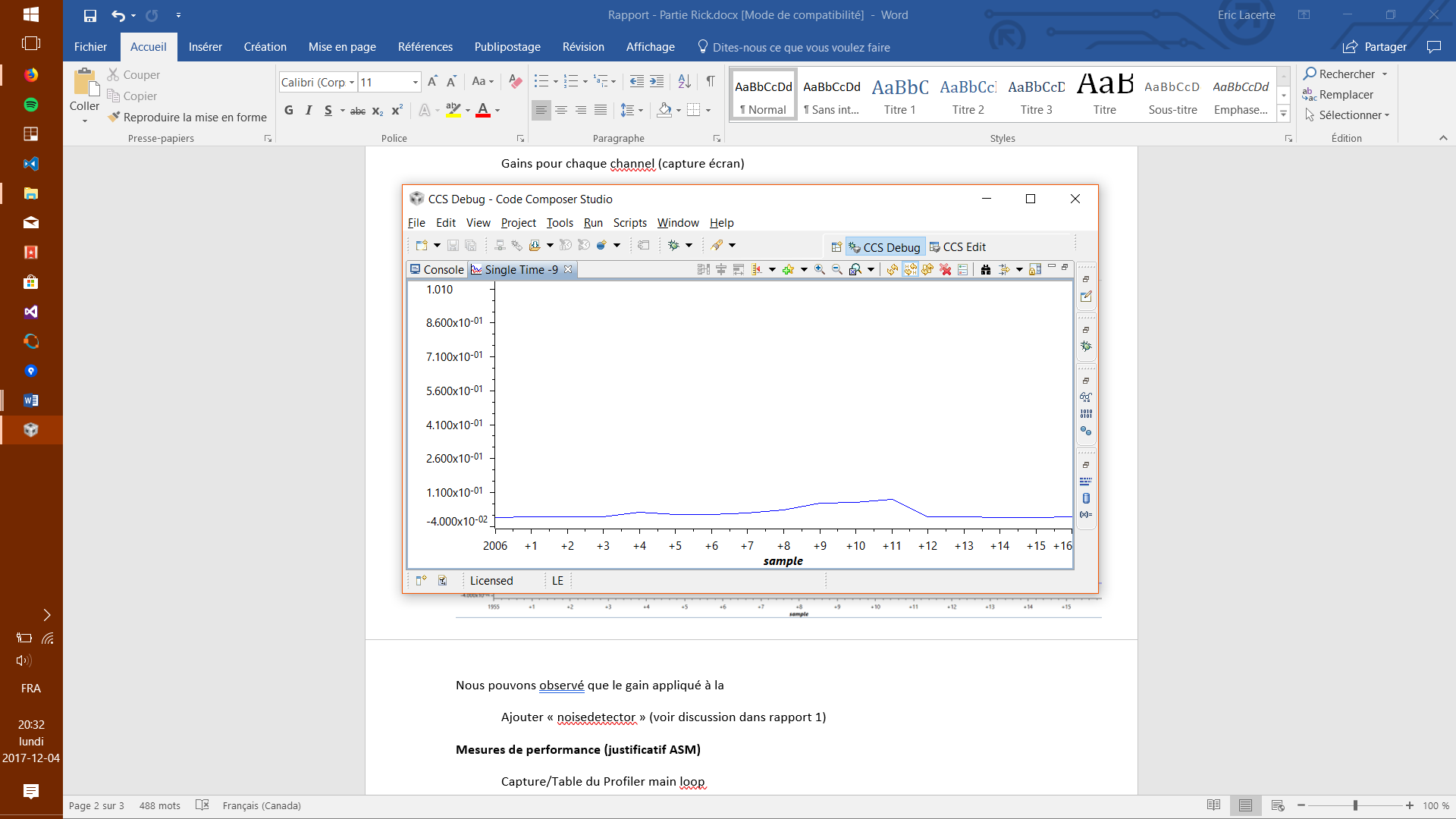


Figure 1 Valeurs de gains calculées par canaux lorsqu'aucune voix n'est présente

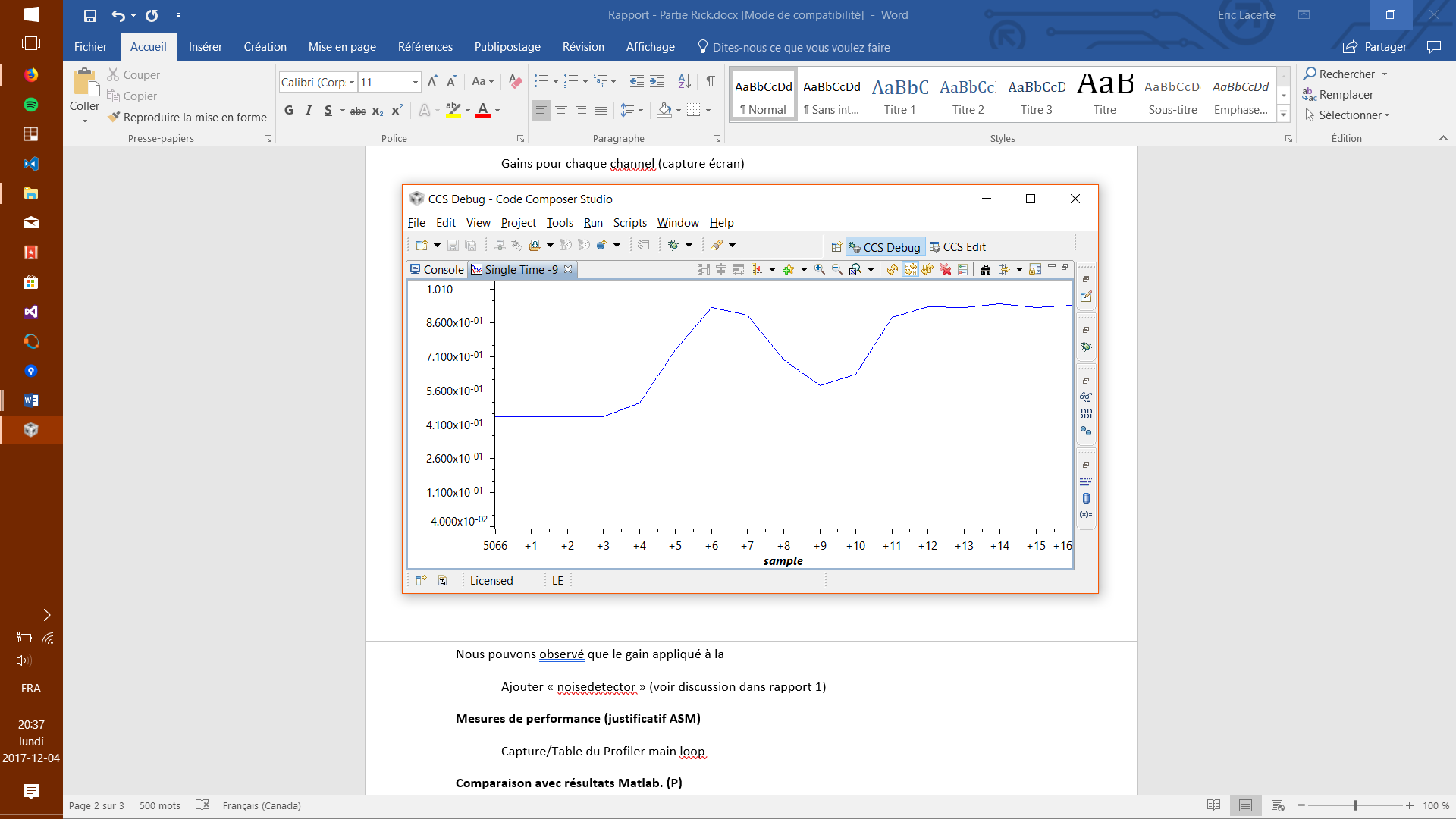


Figure 2 Valeurs de gains calculées par canaux lors de la présence de voix

Nous pouvons constater que le gain appliqué est trop faible lorsque la voix n’est pas présente et que la valeur du gain augmente lorsqu’il détecte la voix. L’algorithme adapte le gain afin d’affaiblir les trames de bruits.

En comparant les gains appliqués entre deux trames consécutives, nous pouvons également observer que la variation du gain appliqué se fait tout en douceur.

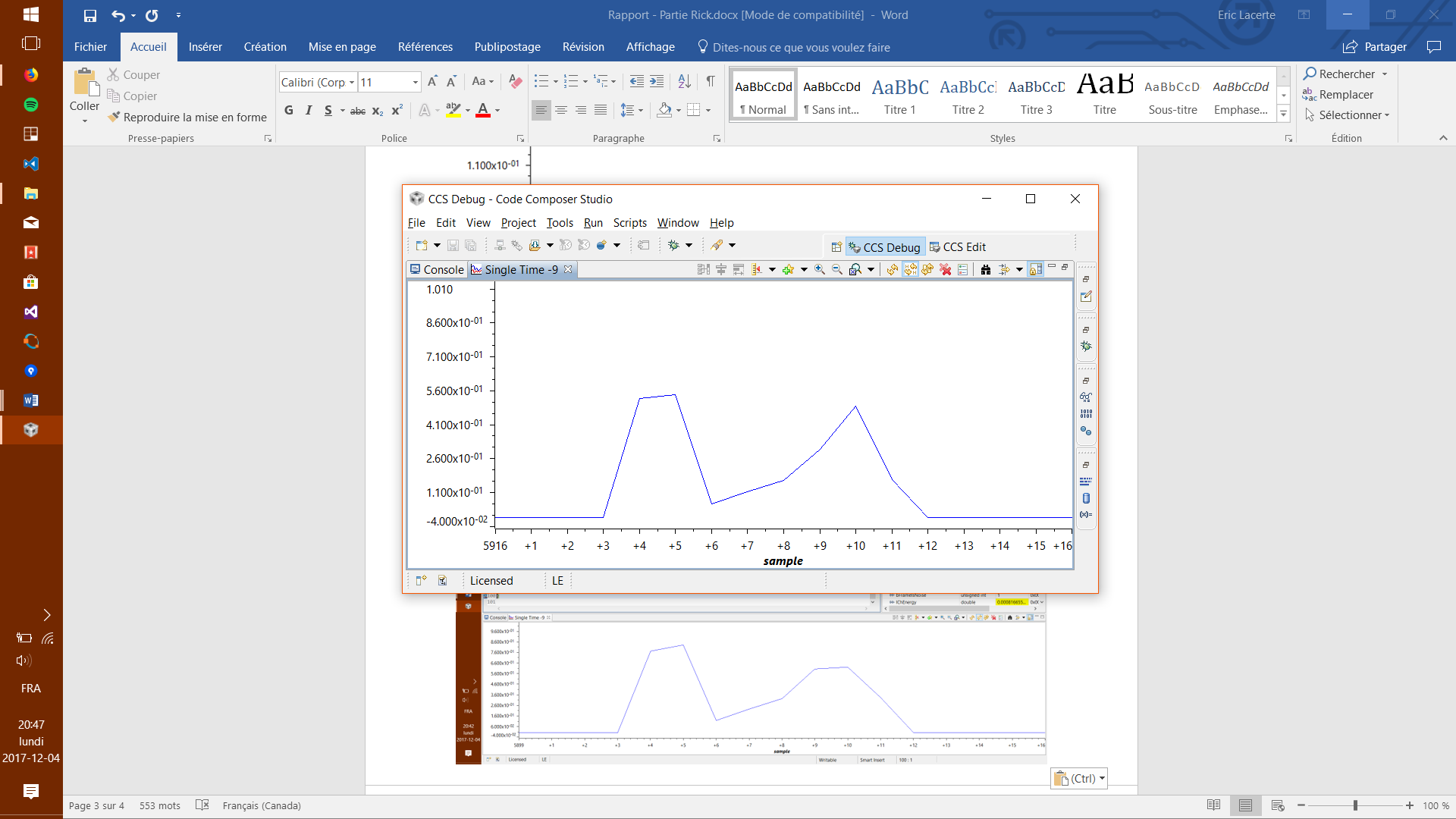


Figure 3 Valeurs de gains appliquées par canaux à la trame n.

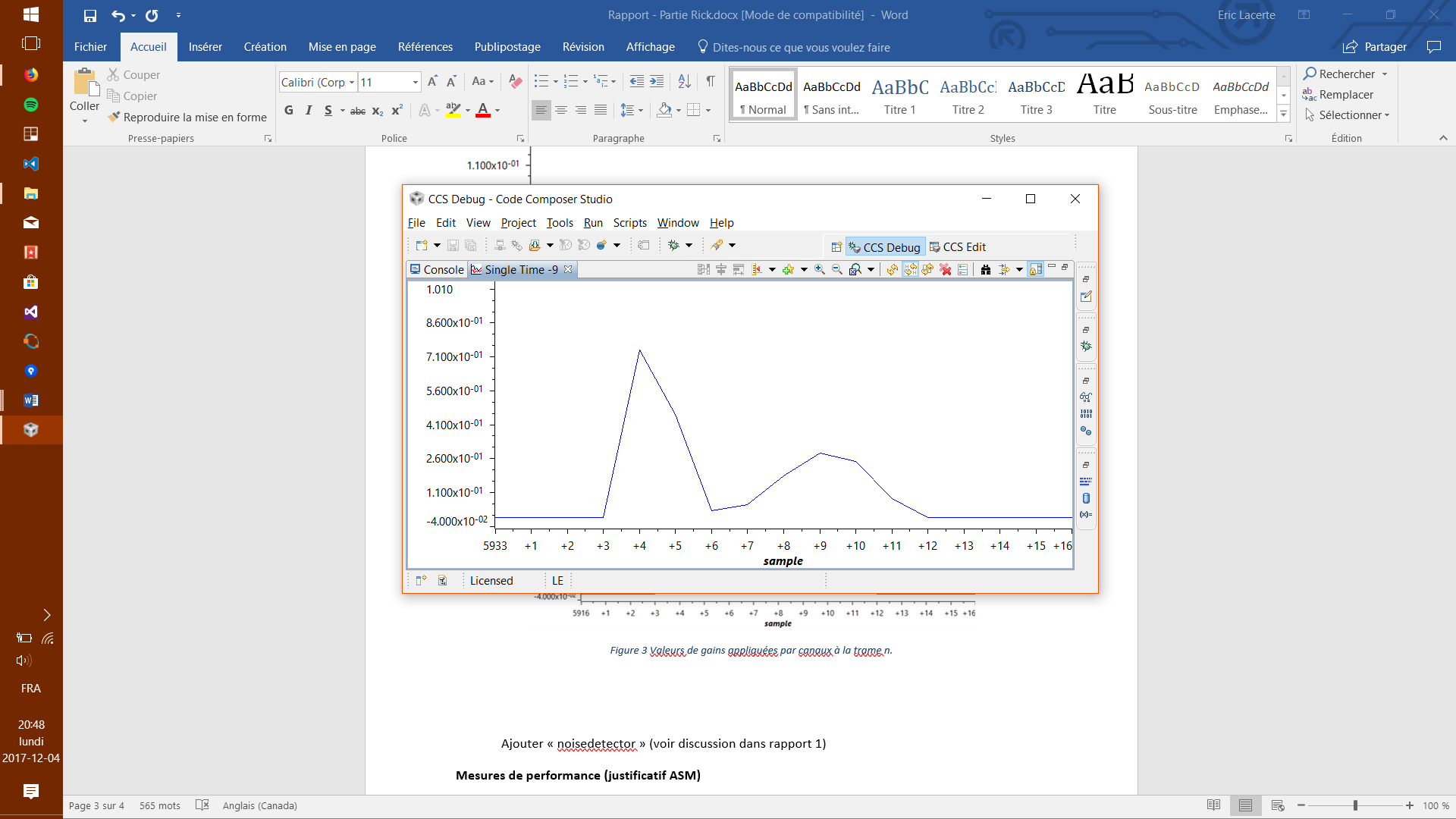


Figure 4 Valeurs de gains appliquées par canaux à la trame n + 1.

Nous avons cependant éprouvé quelques difficultés au niveau du seuil de détection du bruit. Tel que mentionné dans le précédent rapport[[3]](#footnote-3), la mise à l’échelle de l’énergie ne semble pas être adéquate afin d’utiliser l’algorithme du détecteur de bruit (NoiseDetector). Nous avons donc opté pour un seuil de détection du bruit fixe (UINT16\_MAX). En utilisant cette méthode nous pouvons discerner adéquatement les trames contenant seulement du bruit.

## Performances

À l’aide de l’outil de profilage de CCS, nous avons mesurer les nombres de cycles requis pour chacune des fonctions de l’algorithme.

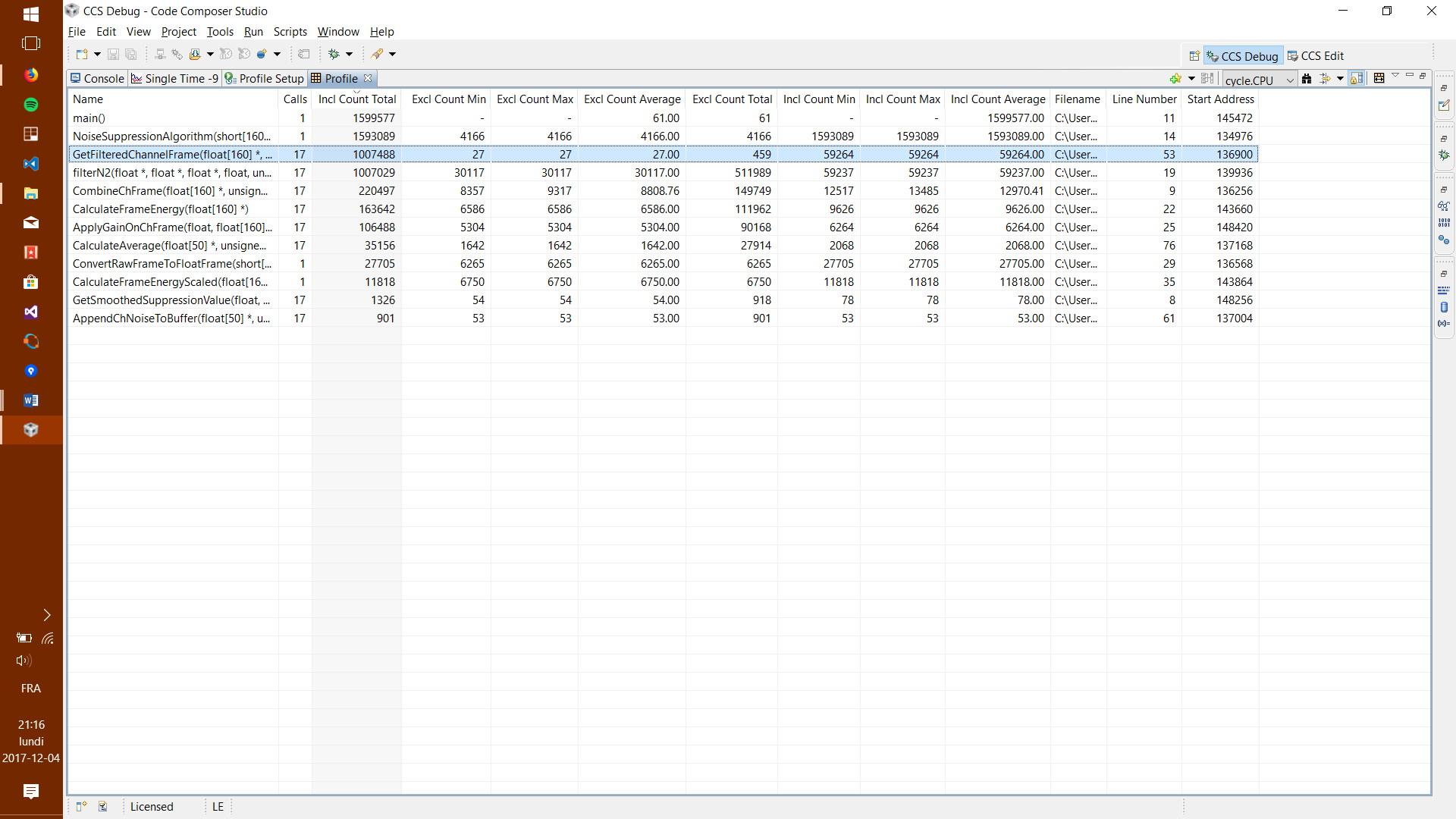


Figure 5 Nombres de cycles requis par fonctions

Nous pouvons constater que la fonction *GetFilteredChannelFrame*  (qui fait appel à la fonction *filterN2*) est la fonctions la plus coûteuse en terme de ressources. Ceci justifie l’optimisation de cette fonction *filterN2* en assembleur tel que discutée précédemment.

## Carte DSK6713 (temps réel)

Lors de l’exécution de l’algorithme sur la carte DSK6713 en temps réel, nous avons éprouvé quelques problèmes. Une distorsion apparaît lorsque l’on utilise tous les 17 canaux. Nous avons dû diminuer le nombre de canaux utilisés pour obtenir un signal audible. Malgré tout, les interruptions semblaient produire les trames de 160 échantillons adéquatement.

# Analyse

Les résultats obtenus sur le simulateur et la carte avec le filtre passe-bande Butterworth d’ordre 2 diffèrent de ceux obtenus avec Matlab. Nous avons généré des fichiers avec le simulateur en appliquant le filtre avec 4, 8 et 17 bandes de notre échelle de fréquences. Nous remarquons que plus nous traitons de bandes avec le filtre, plus le signal semble affecté, jusqu’au point de devenir inintelligible lorsque toutes les 17 bandes de fréquence sont traitées.

C’est peut-être dû à une accumulation d’erreurs (le processeur utilise des données en point flottant simple précision alors que Matlab est en double précision). Une autre piste serait une erreur dans la conception du filtre. Nous avons réutilisé le code du site de l’université de York[[4]](#footnote-4) en y remplaçant les coefficients et le gain par ceux calculés avec Matlab.

La détection du niveau de bruit diverge aussi des résultats obtenus avec Matlab. Pour le même buffer audio, Matlab détermine que 50% des trames ne contienne que du bruit (sans voix) tandis que le code DSP détermine que 90% des trames ne contiennent que du bruit. Bien que l’implémentation soit identique à celle de Matlab et que nous observions une distribution d’énergie semblable à celle décrite dans [1] (voir figures 1 et 2), il y a peut-être encore des problèmes d’échelle (taille des données d’échantillon) tel que discuté dans le premier rapport de ce projet.



Figure 6 distribution cumulative de l’énergie du signal « car.wav »



Figure 7 histogramme de la distribution de l’énergie du signal « car.wav ».

# Conclusions et Recommendations

Suite à ce projet, nous avons expérimenté les limites et les principales difficultés à l’implémentation un algorithme en temps réel. Bien que la fonctionnalité fût démontrée par simulation, l’implémentation de celle-ci requiert des optimisations particulières.

De plus, nous aurions pu générer une librairie commune de l’algorithme qui aurait facilité l’intégration et la validation d’un projet à un autre (projet simulation et temps réel)

# Références

[1] Robert J. McAulay and Marilyn L. Malpass, (1980) SPEECH ENHANCEMENT USING A SOFT-DECISION NOISE SUPPRESSION FILTER

[2] ERIC LACERTE et PHILIPPE LAVOIE, Rapport première étape de projet.

[3] Site web de Department of Computer Science de l’université de York <http://www-users.cs.york.ac.uk/~fisher/mkfilter/trad.html>

[4] Site de Signalogic <http://www.signalogic.com/index.pl?page=codec_samples>

[5] Site du cours SYS835, https://ena.etsmtl.ca/mod/folder/view.php?id=316910

1. Site de signalogic <http://www.signalogic.com/index.pl?page=codec_samples> [↑](#footnote-ref-1)
2. Site du cours SYS835, https://ena.etsmtl.ca/mod/folder/view.php?id=316910 [↑](#footnote-ref-2)
3. ERIC LACERTE et PHILIPPE LAVOIE, Rapport première étape de projet. [↑](#footnote-ref-3)
4. Site web de Department of Computer Science de l’université de York <http://www-users.cs.york.ac.uk/~fisher/mkfilter/trad.html> [↑](#footnote-ref-4)